Greedy Algorithms

The content of this presentation is a collection from various books and online resources, such as Introduction to Algorithms, by CLRS and the lecture notes of Dr. Z. Butler, Dr. Y. Tao and others. Thanks to all for their valuable contributions.

Greedy Algorithms

- Greedy strategy work well for different optimization problems with following characteristics:
  - **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
  - **Optimal substructure:** An optimal solution to the problem contains an optimal solution to sub-problems.
- The second property may make greedy algorithms look like dynamic programming.
- However, the two techniques are quite different.

Greedy vs Dynamic Programming

- Optimal substructure property exploited by both Greedy and DP strategies
- Greedy Choice Property: A sequence of locally optimal choices ⇒ an optimal solution
  - We make the choice that looks best at the moment.
  - Then solve the sub-problem arising after the choice is made
- DP: We also make a choice/decision at each step, but the choice may depend on the optimal solutions to sub-problems
- Greedy: The choice may depend on the choices made so far, but it cannot depend on any future choices or on the solutions to sub-problems

Elements of Greedy Strategy

- DP is a bottom-up strategy
- Greedy is a top-down strategy
  - Each greedy choice in the sequence iteratively reduces each problem to a similar but smaller problem.

- How can you judge whether a greedy algorithm will solve a particular optimization problem?
- Two key ingredients
  - Greedy choice property
  - Optimal substructure property

Key Ingredients of Greedy Strategy

- Greedy Choice Property: A globally optimal solution can be arrived at by making locally optimal (greedy) choices
- In DP, we make a choice at each step but ***the choice may depend on the solutions to subproblems***
- In Greedy Algorithms, we make the choice that seems best at that moment then solve the subproblems arising after the choice is made
  - The choice may depend on choices so far, but it cannot depend on any future choice or on the solutions to subproblems.
- We must prove that a greedy choice at each step yields a globally optimal solution

Key Ingredients

- Shows that the solution can be modified so that a greedy choice made as the first step reduces the problem to a similar but smaller sub-problem.
- Then induction is applied to show that a greedy choice can be used at each step
- Hence, this induction proof reduces the proof of correctness to demonstrating that an optimal solution must exhibit optimal substructure property.
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems

Greedy Algorithms

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current sub-solution.
  - Making the best choice locally at each step, without regard for future consequence,
- Greedy algorithms do not always yield optimal solutions,

  Local optimum ⇒ Global optimum ? ☺
- But, for many problems they do.
- Examples:
  - Dijkstra's shortest path algorithm
  - Prim/Kruskal's MST algorithms
  - …
- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.

Activity-Selection Problem

Activity-Selection Problem

- **Problem Statement**
- Let $S = \{1, 2, \ldots, n\}$ be the set of activities that compete for a resource.
- Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i, f_i)$.
- No two activities can share the resource at any time point.
- We say that activities i and j are compatible if their time periods are disjoint.
  - That is, two activities a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.
- The activity-selection problem is the problem of selecting the largest set of mutually compatible activities.

Activity-Selection Problem

- **Algorithm:**
  - Sort the activities by finish time. Initialize S_m as 0.
  - Take the activity with the earliest finish time in S and copy it to S_m.
  - Delete this activity and all activities that are not compatible with it from S.
  - Keep doing this until S is empty, and then return S_m.

- The time complexity is dominated by the cost of sorting the n items by finish time, which is $O(nlogn)$.

Activity-Selection Problem

- **Example:**
- Consider 11 activities, with corresponding start time (s_i) and finish time (f_i), shown in the table.
- What is the output of the greedy algorithm, discussed above?

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

- **Solution:**
- A = {1, 4, 8, 11} is an optimal (why?) solution, obtained by using the greedy activity-selection algorithm.
- Again, A = {2, 4, 9, 11} is another optimal solution.
- There can be others as well…
- But, here we are trying to find a solution (not all).

Activity-Selection Problem

- The greedy choice in the above algorithm is:
  - To pick the compatible activity with the earliest finish time.

- Why?
  - Intuitively, this choice leaves as much opportunity as possible for the remaining activities to be scheduled.
  - That is, the greedy choice maximizes the amount of unscheduled time remaining.

Activity-Selection Problem: Optimality Proof

- The big question here is:
- Does the greedy algorithm give us an optimal solution?
- The answer is YES.
- We can prove this as follows.

Activity-Selection Problem: Optimality Proof

- **Theorem-1:** Prove that the greedy Activity-Selection algorithm returns an optimal solution.
  - **Recall:** *Greedy Choice Property* - At least one optimal solution contains the greedy choice

- To show this:
  - Let A be the set of activities selected by the greedy algorithm
  - Consider any non-overlapping set of activities B.
    - That is, we considered an arbitrary, non-overlapping set of activities, B and which is assumed to be an optimal solution (obtained by using some other method).
  - We will show that $|A| \geq |B|$ by showing that we can replace each activity in B with an activity in A.
    - Here, we are trying to prove that the number of activities in A is $\geq$ that in B.
  - This will show that A has at least as many activities as any other non-overlapping schedule and thus that A is optimal.

Activity-Selection Problem: Optimality Proof

- Let a_x be the first activity in A that is different than an activity in B.
- Then
  - $A = a_1, a_2, \ldots, a_x, a_{x+1}, \ldots$ and
  - $B = a_1, a_2, \ldots, b_x, b_x + 1, \ldots$
- But since A was chosen by the greedy algorithm, a_x must have a finish time which is earlier than the finish time of b_x.
- Thus, $B' = a_1, a_2, \ldots, a_x, b_{x+1}, \ldots$ is also a valid schedule. That is, $(B' = B - \{b_x\} \cup \{a_x\})$
- Continuing this process, we see that we can replace each activity in B with an activity in A.

Therefore, we conclude that there always exists an optimal solution that begins with a greedy choice.

Activity-selection problem

- **Theorem-2**: Prove that, if A is an optimal solution for S, then $A' = A - \{a_1\}$ is an optimal solution to $S' = \{i \in S, s[i] \geq f[a_1]\}$. What is a_1
  - **Recall:** *Optimal Substructure* - An optimal solution can be made from the greedy choice plus an optimal solution to the remaining sub-problem.

Activity-selection problem

- Proof by contradiction:
- Let *S* be an optimal solution to the problem, which contains the greedy choice
- Consider $S' = S - \{a_1\}$. S' is not an optimal solution to the problem of selecting activities that do not conflict with a_1
- Let S'' be an optimal solution to the sub-problem of picking activities that do not conflict with a_1.
- Consider $S''' = S'' \cup \{a_1\}$. S''' is a valid solution to the problem, $|S'''| = |S''| + 1 > |S'| + 1 = |S|$ (since S' is not optimal).
- This contradicts our assumption that S is an optimal solution.

Knapsack Problem

Knapsack Problem

- Next we discuss a version of Knapsack Problem, called Fractional Knapsack Problem, and see a greedy algorithm for the same.
- This greedy solution doesn't work for the 0/1 knapsack (which must be solved using Dynamic Programing).
  - The 0/1 knapsack problem is computationally hard problem.

- **0/1 knapsack problem:** Consider the situation when a burglar breaks into a house. After seeing all items in the house, he has a clear idea of what the value of each item is and what the volume of each item is.
  - Now, he wants to take everything but he only has a "knapsack" of a certain size.
  - Thus, he wants to take a subset of items of maximum total value but still fits his knapsack.
  - In addition, all items are indivisible (i.e. he cannot take half of an item and get half its value). What should he take?
  - 0/1 means "Take-it or Leave-it". This is also called Integer Knapsack Problem.

Knapsack Problem

- **Problem Definition (0/1 knapsack problem):**
- In a knapsack problem, there is a set I containing n items, labeled $1,2, \dots, n$.
- Each item is associated with a value $v_1, v_2, \dots, v_n$ and a weight $w_1, w_2, \dots, w_n$.
- In addition, there is a constraint W (which is the capacity of the Knapsack).
- The problem asks to find the subset $I' \subseteq I$ that **maximizes** the total value of items in it $\sum_{i \in I'} v_i$ subject to the constraint $\sum_{i \in I'} w_i \leq W$.

Fractional Knapsack Problem

- What happens if the items are divisible. For example, if the burglar can take half a bag of item-1 and get half of its value. Still having the constraint of the total size of his knapsack, what should be his strategy to take items?

- **Problem Definition (Fractional Knapsack Problem):**
- In a Fractional Knapsack Problem, there is a set I containing n items, labeled $1,2, \dots, n$.
- Each item is associated with a value $v_1, v_2, \dots, v_n$ and a weight $w_1, w_2, \dots, w_n$.
- In addition, there is a constraint W (which is the capacity of the Knapsack).
- The problem asks to find the weight of each item to take $w_1', w_2', \dots, w_n'$ that maximizes the total value of taken items $\sum_{i=1}^{n} v_i \frac{w_i'}{w_i}$ subject to the constraint $\sum_{i=1}^{n} w_i' \leq W$ and $w_i' \leq w_i$.

Knapsack Problem

- **Example**
- Given a knapsack of capacity $W = 50$ and three items, each with weights and values, as shown in the table.
- 0/1 knapsack problem: Max. possible value = 220, by taking items of weight 20 and 30.
- Fractional Knapsack: Since now we are allowed to pick fractions, maximum possible value = 240. By taking full of 10kg, 20Kg and 2/3 of last item.
  - That is: Max Value $= 60 + 100 + \frac{2}{3} * 120 = 240$
  - Weight $= 10 + 20 + \frac{2}{3} * 30 = 50 = W$

Items (i)	1	2	3
Weight (w_i)	10	20	30
Value (v_i)	60	100	120
$\frac{v_i}{w_i}$	6	5	4

Q. What will happen is W = 55?

Greedy solution for Fractional Knapsack

Items (i)	1	2	3
Weight (w_i)	10	20	30
Value (v_i)	60	100	120
$\frac{v_i}{w_i}$	6	5	4

- **Algorithm:**
- Calculate $\frac{v_j}{w_j}$ for each $i = 1\ to\ n$
- Sort items by decreasing order of $\frac{v_j}{w_j}$. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding weights be w_i
- Let k be the current weight limit (Initially, $k = W$). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, we take item i, and $k = k - w_i$, then consider the next unselected item.
  - Else if $k < w_i$, we take a fraction f of item i, i.e., we only take $f = \frac{k}{w_i}$ (< 1) of item i, which weights exactly k.

Fractional Knapsack Problem

- The time complexity is dominated by the cost of sorting the n items by value per unit weight, which is $O(n \log n)$.
  - Notice that it takes $O(n \log n)$ for sorting and O(n) time to traverse and pick from the list of items until the knapsack is full.

- We claim that the total cost for this set of items is an optimal cost.

Fractional Knapsack Problem

- Greedy Choice Property: Let j be the item with maximum $\frac{v_i}{w_i}$. Then there exists an optimal solution in which you take as much of item j as possible.
  - That is, there exists an optimal solution that begins with the greedy choice given above.
- Proof: (by contradiction)
- Suppose that there exists an optimal solution that didn't take as much of item j as possible.
- If the knapsack is full, then there must exist some item $k \neq j$ with $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ that is in the knapsack.
- Thus, not all of item j is in the knapsack, we can a piece of k, of ϵ weight, out of the knapsack, and put a piece of j with ϵ weight in.
- This increases the knapsacks value by

$$\epsilon \frac{v_j}{w_j} - \epsilon \frac{v_k}{w_k} > 0$$

- This contradiction our assumption that an optimal solution can be formed without taking as much as of item j as possible.

Fractional Knapsack Problem

- The Fractional Knapsack Problem exhibits Optimal Substructure.
- That is, given the problem S with an optimal solution X with value V, we want to prove that the solution $X' = X - x_j$ is optimal to the problem $S' = S - \{j\}$ and the knapsack capacity $W' = W - x_j$.
- Proof by contradiction:
  - Assume that X' is not optimal to S'
  - There is another solution X'' to S' that has a higher total value $V'' > V'$
  - Then $X'' \cup \{x_j\}$ is a solution to S with value $V'' + x_j > V' + x_j > V$
  - This is a contradiction as V is the optimal value.

Greedy Algorithm II

Huffman Codes

Huffman Codes

- Huffman codes provide a method of encoding data efficiently.
- Normally when characters are coded using standard codes like ASCII or the Unicode, each character is represented by a fixed-length codeword of bits (e.g., 8 or 16 bits per character).
- Fixed-length codes are popular, because its is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing.
- However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

Huffman Codes

- Suppose that we have an alphabet Σ (like the English alphabet).
- The goal of coding is to map each alphabet to a binary string - called a codeword - so that they can be transmitted electronically.
- For example, suppose $\Sigma = \{a, b, c, d, e, f\}$. Assume that we agree on
  - a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101.
- Then, a string such as "bed" can be encoded as 001100011.
- The above is an example of a fixed-length codes, where can easily see that if we have a file of 1000 characters (selected from Σ): then the corresponding fixed-length encoding will have 3000 bits.
- Decoding: Simply chop the data into blocks of 3 bits each and decode one symbol for each block.
  - Ease of decoding is an advantage of fixed length codes
- The question is:
  - Can we do better, in terms of the average number of bits needed per symbol?
  - What about variable length encoding?

Huffman Codes

- We can achieve better coding efficiency (with less number of bits), if the frequencies of the letters are known.
- In fact, the frequencies (in percent %) of English single letters in text has been studied, as shown in the following table.
  - Here we can see the letter ‘e’ appears most frequently at 12.1%, followed by ‘a’ and ‘t’ at about 8.5%, followed by other characters.

A : 8.55	K : 0.81	U : 2.68
B : 1.60	L : 4.21	V : 1.06
C : 3.16	M : 2.53	W : 1.83
D : 3.87	N : 7.17	X : 0.19
E : 12.10	O : 7.47	Y : 1.72
F : 2.18	P : 2.07	Z : 0.11
G : 2.09	Q : 0.10	
H : 4.96	R : 6.33	
I : 7.33	S : 6.73	
J : 0.22	T : 8.94	

Huffman Codes

- **Variable length encoding:** The intuition is to assign shorter codewords to more frequent symbols, longer codewords to less frequent symbols.

- **Example:**
- If we know that the frequencies of $\{a, b, c, d, e, f\}$ are $\{0.1, 0.2, 0.13, 0.09, 0.4, 0.08\}$, respectively.
- The variable length encoding of the characters is shown in the table in the gray column.
- Example: We can encode the string “bed” as 11101101.

- With this encoding scheme: we can see that the total number of bits required for encoding 1000 characters is 2370 bits (See the table for detail).

$\sum$	Variable length encoding (bits)	Frequency (f_i)	Chars out of 1000 $c_i = (f_i \times 1000)$	Bits per char. (b_i)	Total bits $(c_i \times b_i)$
a	100	0.1	100	3	300
b	111	0.2	200	3	600
c	101	0.13	130	3	390
d	1101	0.09	90	4	360
e	0	0.4	400	1	400
f	1100	0.08	80	4	320
Total:			1000		2370

Huffman Codes

- Consider the following example.

$\sum$	Probabilities	Code 1	Code 2	Code 3
a	0.60	00	0	0
b	0.30	01	1	10
c	0.05	10	10	110
d	0.05	11	11	111

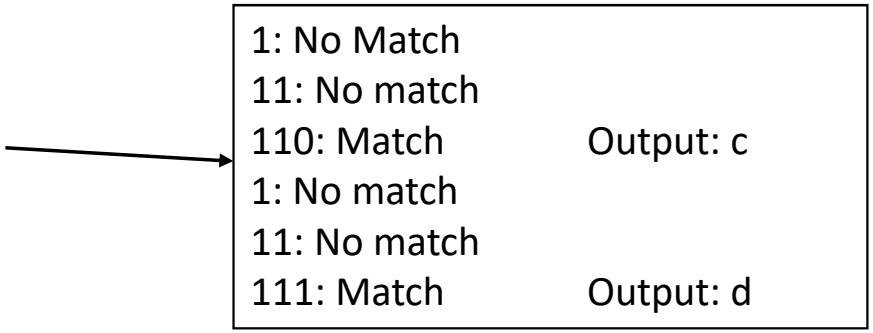
- Code-1 is of fixed length. Whereas, code-2 and code-3 are of variable length codewords.
- However, what's the problem with code-2?
  - Its in Decoding.
  - Suppose we receive the encoding: 110111.
  - The decode can think it as "BBABBB", or "BCDB" or "DADB", or …
  - Code-2 is not uniquely decodable.

- Do we have the same problem with code-3? Why?
- What is the difference between code-2 and code-3?

Huffman Codes

- Code-3 is a prefix-free variable length code
- That is, no codeword is the prefix of any other codeword.
- Such prefix-free codes are uniquely decodable. For example: we can decode 110111 as “cd”.
  - For this, we scan 110111 from left to right, and match it with the codewords.
- Thus, our algorithm must generate prefix free codewords by considering the frequencies of the characters.

$\sum$	Probabilities	Code 1	Code 2	Code 3
a	0.60	00	0	0
b	0.30	01	1	10
c	0.05	10	10	110
d	0.05	11	11	111



Huffman Codes

$\sum$	Probabilities	Code 1	Code 2	Code 3
a	0.60	00	0	0
b	0.30	01	1	10
c	0.05	10	10	110
d	0.05	11	11	111

- What is the efficiency of fixed length (Code I) versus prefix-code III?
- The code-1, which is a fixed length code requires 2 bits per symbol.
- Where as code-3 requires:

$$0.6 \times 1 \; + \; 0.3 \times 2 \; + \; 0.05 \times 3 \; + \; 0.05 \times 3 \; = \; 1.5 \text{ bits per symbol}$$

- Therefore, code-3 is expected to save (in comparison to code-1):

$$1 - \frac{1.5}{2} = 25\% \text{ of bits!}$$

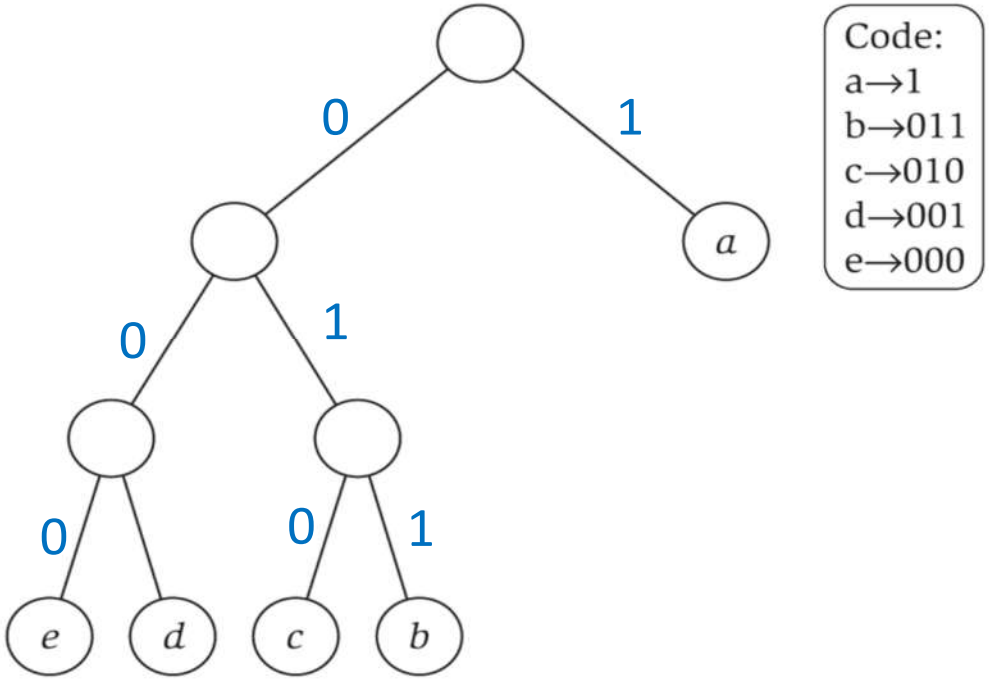
Huffman Codes

- **Optimal Prefix Codes:**
- Among all possible prefix codes, can we devise an algorithm that will give us an optimal prefix code?
  - One that most efficiently encodes the symbols with the lowest average bits per symbol
- Huffman codes are optimal prefix codes

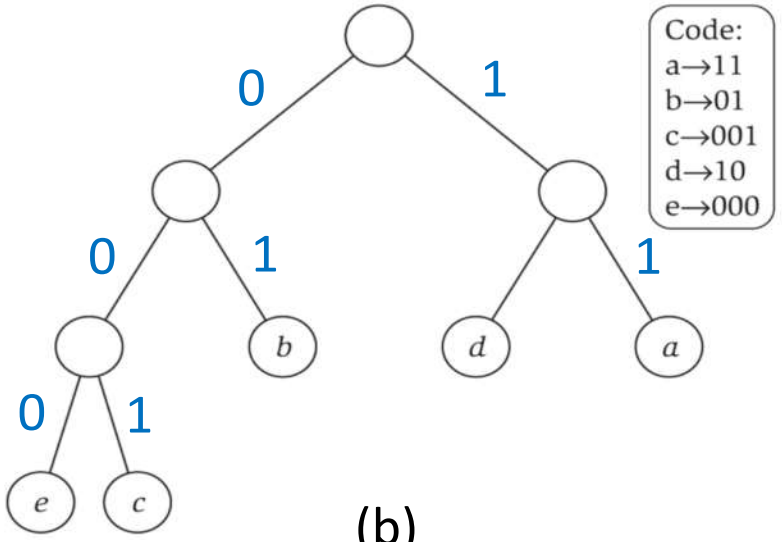
Huffman Codes

- We can represent prefix-free codes by a binary trees.
- For this, let each leaf node represents a symbol
- Each path (from root to the symbol) represents an encoding for that symbol.
  - Traveling to the left child is a ‘0’
  - Traveling to the right child is a ‘1’
- Why is it a prefix code?
  - For a symbol to a prefix of another, it would have to be a non-leaf.
  - Since the paths of any two symbol branches-off into two different sub-trees, the codeword of a symbol can not be the prefix of any other codeword.
- See the following examples.

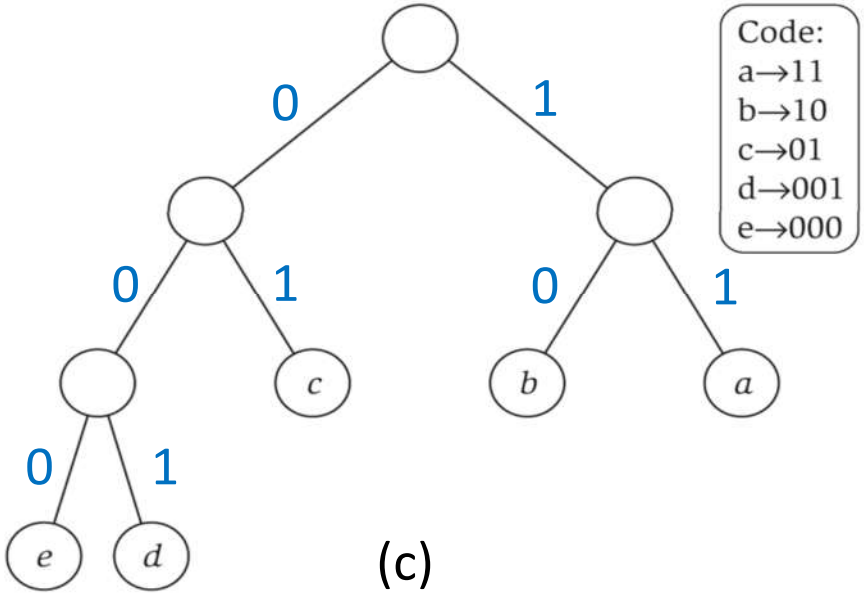
Huffman Codes



(a)



(b)



(c)

- The figure (a), (b) and (c) depict three different prefix-free code for the alphabet S={a,b,c,d,e}.
- Why must every letter be at the leaf? (Hint: prefix free)

Huffman Codes

The Prefix Coding Problem: Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$ find a binary prefix code C for A that minimizes the number of bits, i.e.,

$$B(C) = \sum_{i=1}^{n} f(a_i) L(c(a_i))$$

needed to encode a message of $\sum_{i=1}^{n} f(a_i)$ characters, where $c(a_i)$ is the codeword for encoding a_i, and $L\big(c(a_i)\big)$ is the length of the codeword $c(a_i)$.

- **Question:** Compute B(C) for the three prefix-free tree shown before. The frequencies of the symbols are given in the following table.
- Which one of the three is a good prefix free code? Is this the best?

i	(f_i)	$L\big(c(a_i)\big)$ for Fig.-(a)	$L\big(c(a_i)\big)$ for Fig.-(b)	$L\big(c(a_i)\big)$ for Fig.-(c)
a	0.10	1	2	2
b	0.27	3	2	2
c	0.13	3	3	2
d	0.09	3	2	3
e	0.41	3	3	3

Huffman Codes

- Huffman developed a greedy algorithm for solving this problem and producing a minimum cost (optimum) prefix code. The code that it produces is called a Huffman code.
- **Algorithm:**
- Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea)
  - Label the root of this subtree as z.
  - The idea is to minimize $B(C) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$ at each step.
- Step 2: Set frequency $f(z) = f(x) + f(y)$.
  - Remove x, y and add z creating new alphabet $A' = A \cup \{z\} - \{x, y\}$.
  - Note that $|A'| = |A| - 1$.
- Step 3: Repeat this procedure, with new alphabet A', until an alphabet with only one symbol is left.

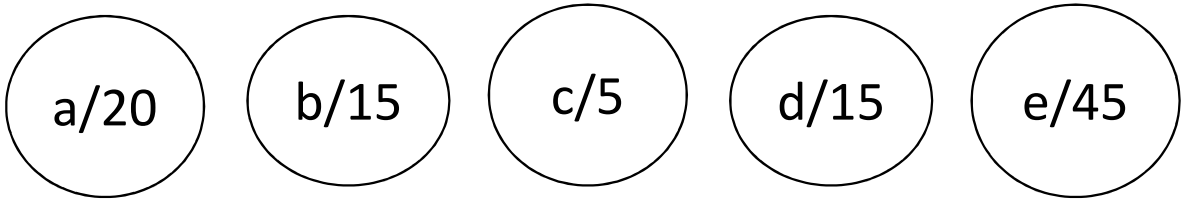
Huffman Codes

- Given an alphabet A with frequency distribution {f(a) : a ∈ A}.
- The binary Huffman tree is constructed using a priority queue, Q, of nodes, with labels (frequencies) as keys.
- Running time is $O(n \log n)$, as each priority queue operation takes time $O(\log n)$

```
Huffman(A)
{   n = |A|;
    Q = A;
    for i = 1 to n − 1
    {   z = new node;
        left[z] =Extract-Min(Q);
        right[z] =Extract-Min(Q);
        f[z] = f[left[z]] + f[right[z]];
        Insert(Q, z);
    }
    return Extract-Min(Q)
}
```

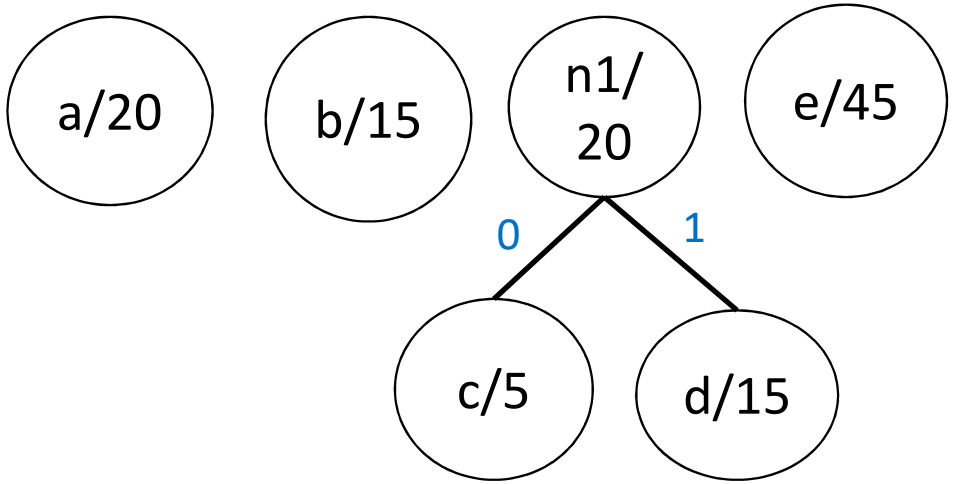
Huffman Codes (Example)

- Let $A = \{a/20, b/15, c/5, d/15, e/45\}$ be the alphabet and its frequency distribution.



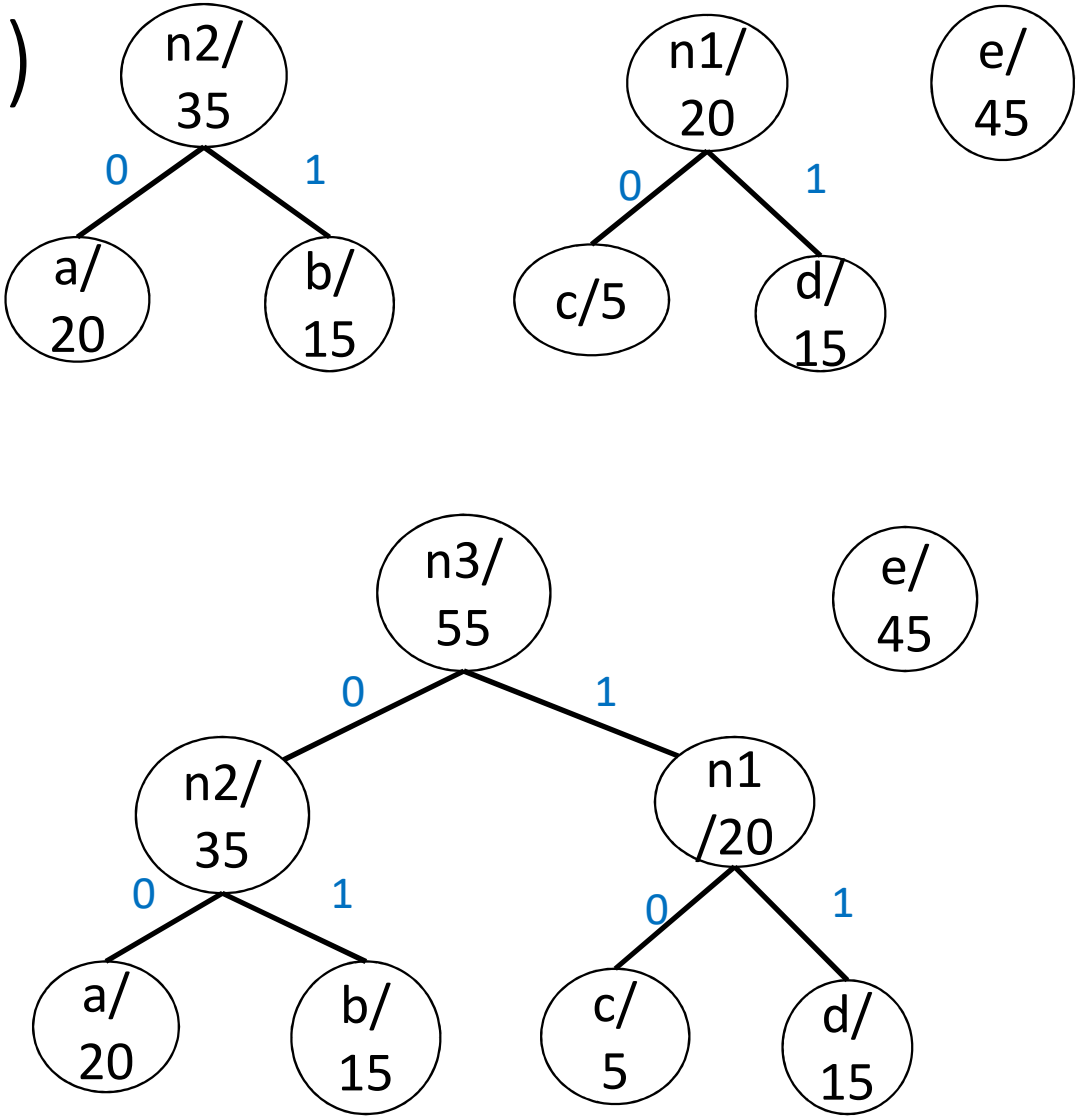
- In the first step of Huffman coding, we merge c and d, by creating a new alphabet (say n1) with frequency 20.
- Alphabet is now

  $A1 = \{a/20, b/15, n1/20, e/45\}.$



Huffman Codes (Example)

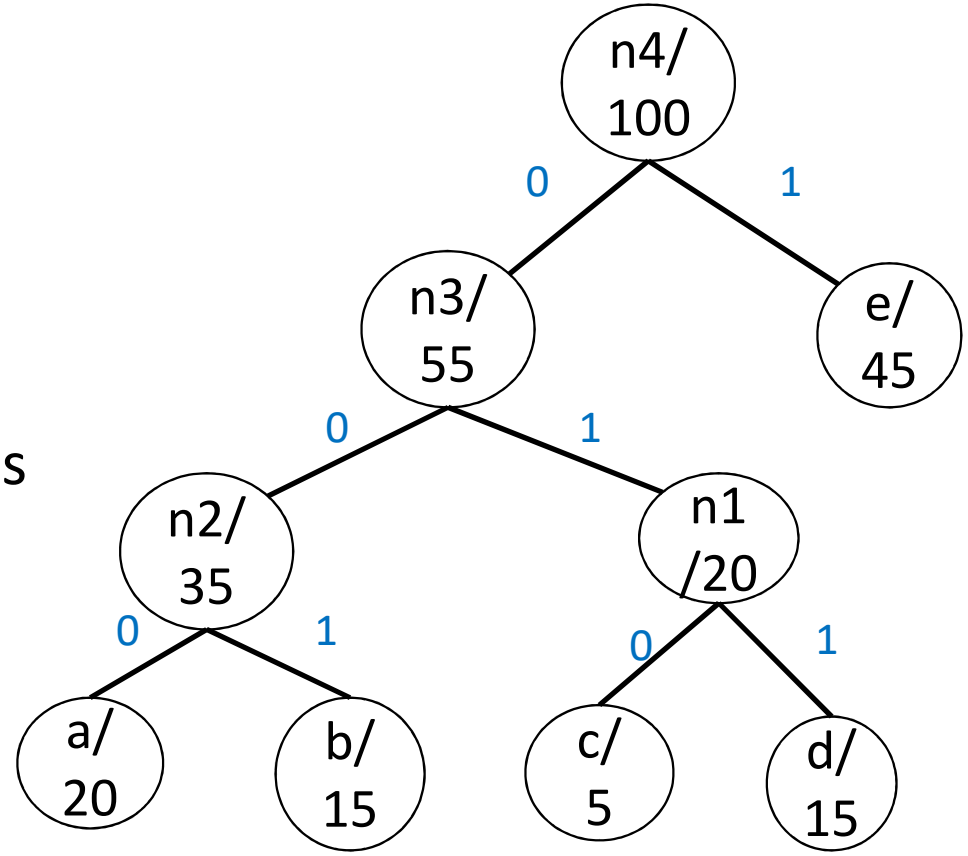
- Alphabet is now $A1 = \{a/20, b/15, n1/20, e/45\}$.
- Next the algorithm merges a and b, as discussed above, to create new alphabet A2={n2/35,n1/20,e/45}.
  - We can also have merged $n1$ and b.
- Then, the algorithm merges n1 and n2.
  - Creating a new alphabet set: $A3 = \{n3/55, e/45\}$.



Huffman Codes (Example)

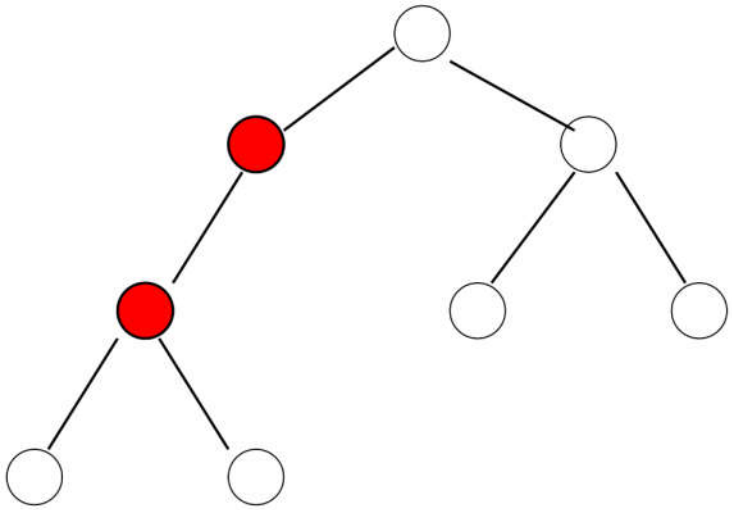
- Finally the algorithm merges e and n3, as shown the figure.
  - We have a single node: $A4 = \{n4/100\}$
  - Which terminates the algorithm.
- Thus, the Huffman code for the example is

  a = 000, b = 001, c = 010, d = 011, e = 1.
- But, is this the optimum (minimum-cost) prefix code for this distribution?



Huffman Codes

- Definition: A binary **tree T is full** if each node is either a leaf or possesses exactly two child nodes



- Lemma: The tree for any optimal prefix code must be “full”, meaning that every internal node has exactly two children.
- Proof:
  - If some internal node had only one child then we could simply get rid of this node and replace it with its unique child.
  - This would decrease the total cost of the encoding.

Huffman Codes

- Lemma: Consider the two letters, x and y with the smallest frequencies. Then prove that there is an optimal code tree, in which these two letters are the sibling leaves in the tree in the lowest level.

…is this a greedy choice property?

- Proof:
  - Here too, the proof is by contradiction.
  - Assuming that T is an optimum prefix-free code tree.
  - If x and y (with smallest frequencies) are not the sibling leaves in the tree in the lowest level, then there must exist other two nodes, say b and c, as siblings and are at the maximum depth (say d) in T. (Why? Because T is full…)
    - Notice that, number of bits used for encoding a character is equal to its depth (d) in the tree.
  - Without loss of generality, let's assume that the left sibling has the smallest frequency. That is, $f(b) \leq f(c)$ and $f(x) \leq f(y)$ (if this is not true, then we can simply rename the characters).

Huffman Codes

- Proof (continues):
  - Since x and y are the symbols with smallest frequencies, it follows that $f(x) \leq f(b)$ (they may be equal) and $f(y) \leq f(c)$ (may be equal).
  - However, since b and c are at the deepest level of the tree, we know that $d(b) \geq d(x)$ and $d(c) \geq d(y)$.

  - Next we switch the position of x and b in the tree, to generate a different tree T'. Comparing the costs of T and T', we get

$$B(T') = B(T) - f(x)d(x) - f(b)d(b) + f(x)d(b) + f(b)d(x)$$
$$. \quad = B(T) - \big(f(b) - f(x)\big)\big(d(b) - d(x)\big)$$

  - Since, $\big(f(b) - f(x)\big)\big(d(b) - d(x)\big) \geq 0$, we have $B(T') \leq B(T)$.
  - Thus, the tree T' is better than T.

  - With similar arguments, we can switch y and c to get a new tree T'', which will be better than T'.

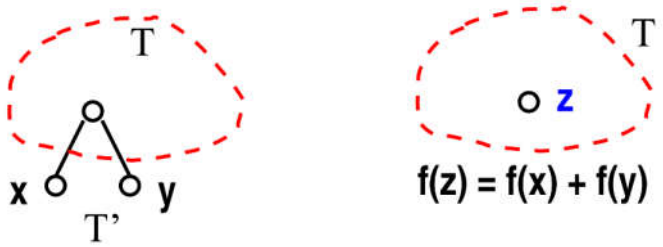
  - Thus, the two letters, x and y with the smallest frequencies, must be the sibling leaves in the tree (T) at the lowest level.

Huffman Codes

- Claim: Huffman's algorithm produces an optimum prefix code tree.
- Proof: By induction on n.
  - When $n = 2$, obvious.
  - Assume inductively, that the Huffman's algorithm produces an optimum tree for strictly fewer than n letters.
  - So, we need to show, this is true when the number of letters is exactly n.

  - Consider an alphabet A' of n letters.
  - Let T' be an optimum tree for A' with the two letters of lowest frequency x and y as sibling leaves (exists by Lemma).
  - Let T be the coding tree for $A = A' \cup \{z\} - \{x, y\}$ ($n - 1$ leaves) obtained by removing x and y and replacing their parent by z.

Huffman Codes



- Proof (continues):
  - That is, $d(x) = d(y) = d(z) - 1$ and $f(z) = f(x) + f(y)$, as shown the figure. So,

$$B(T) = B(T') - f(x)d(x) - f(y)d(y) + f(z)d(z)$$
$$. \quad = B(T') - (f(x) + f(y))$$

  - By the induction hypotheses, the Huffman algorithm gives a Huffman code tree H_A that is optimal for A.
  - Let H_{A1} be the tree obtained by adding x and y as children of z in H_A.
  - Note that, as in the calculations for T and T', we have:

$$B(H_{A1}) = B(H_A) + f(x) + f(y)$$
$$. \quad \leq B(T) + f(x) + f(y) = B(T')$$

  - Since we started with the assumption that T' was an optimal tree for A' this implies that H_{A1}, which is exactly the tree constructed by the Huffman algorithm, is also an optimal tree for A'

Greedy Algorithm

Graph Problems: Minimum Spanning Trees

The content of this presentation is a collection from various books and online resources, such as Introduction to Algorithms, by CLRS and the lecture notes of Dr. K. Schwarz and others. Thanks to all for their valuable contributions.

Greedy Algorithm

- In the following slides, we will see more Greedy algorithms for some graph problems.
- In particular,
  - Kruskal’s Minimum Spanning Tree,
  - Prim’s Minimum Spanning Tree,
  - Dijkstra's algorithm,
  - …

- Let’s start with some definition.

Definitions

- **Definitions**
  - A *tree* is an undirected graph $G = (V, E)$ that is connected and acyclic.
  - An undirected graph is called minimally connected, iff it is connected and removing any edge disconnects it.
  - The following are equivalent statements:
    - G is a tree.
    - G is connected and acyclic.
    - G is minimally connected (removing any edge from G disconnects it.)
    - G is connected and $|E| = |V| - 1$

Spanning Tree

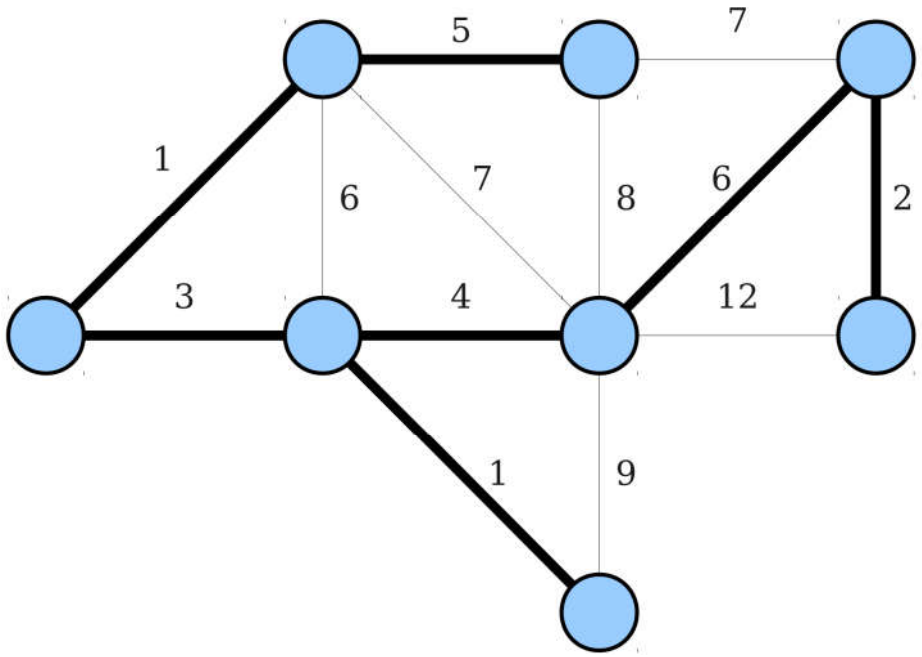
- A *spanning tree* of a graph G is a subset of edges of G that form a tree and include all vertices of G.
- Suppose that each edge $(u, v) \in E$ is assigned weight/cost $c(u, v)$.
- The cost of a tree T, denoted $c(T)$, is the sum of the costs of the edges in T:

$$c(T) = \sum_{(u,v)\in T} \mathrm{c(u,v)}$$

- Finally, the *Minimum Spanning Tree* (MST) problem: Given an undirected graph $G = (V, E)$ and the edge weights, find a spanning tree T^* of minimum weight, i.e., $\min\{c(T)\}$.

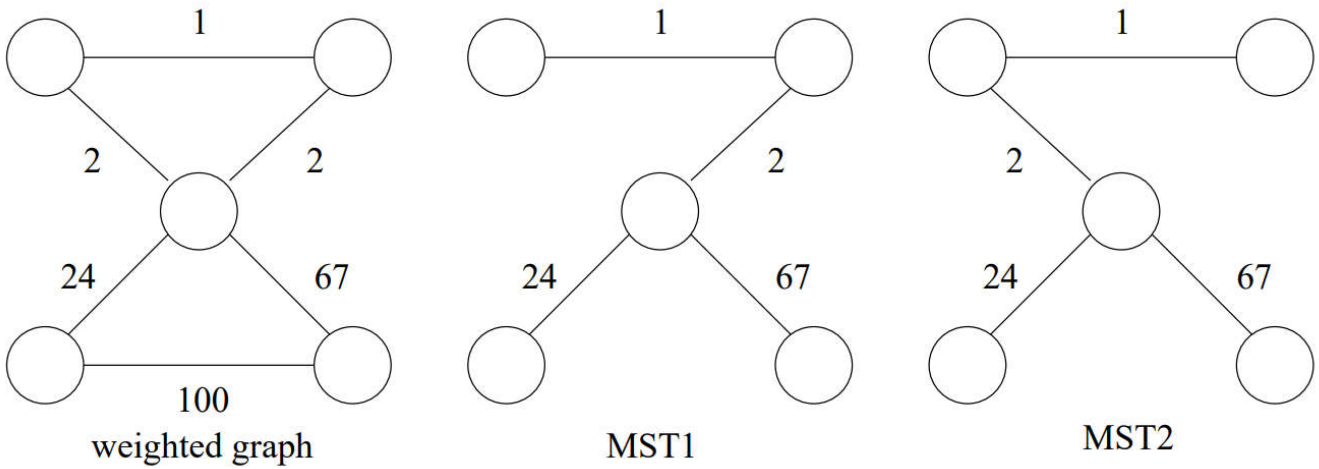
Minimum Spanning Tree (MST)

- Figure shows an example of a spanning tree.
  - Is this a MST, as well?
- There are many greedy algorithms for finding MSTs:
  - Borůvka's algorithm (1926)
  - Kruskal's algorithm (1956)
  - Prim's algorithm (1930, rediscovered 1957).
- Next, we explore the Kruskal's and Prim's algorithms



MST

- Note that the minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique.
- Example:



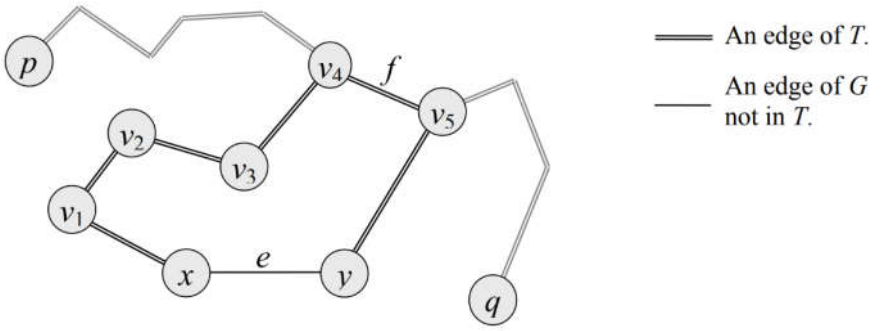
MST

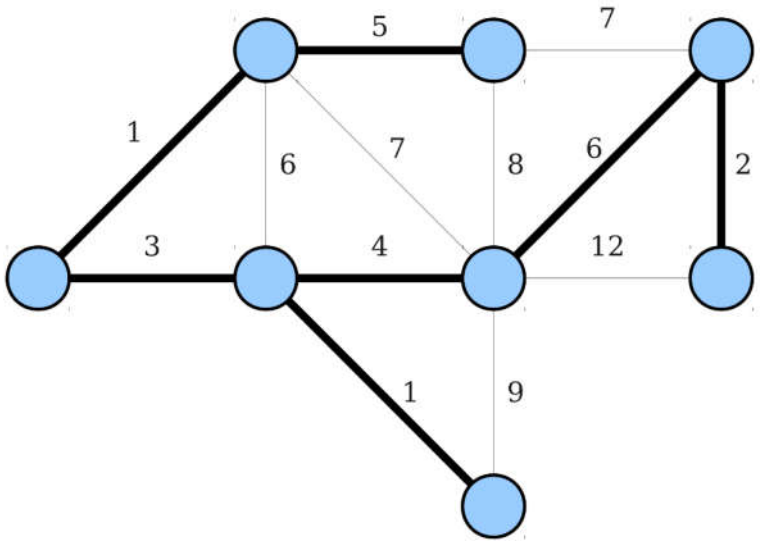
- Lets see a property of MSTs called the cycle property.

- **Lemma (Cycle Property):** If $e = (x, y)$ is an edge in G and is the heaviest edge on some cycle C, then (x, y) does not belong to any MST of G.

- Proof if simple:
  - Let $e = (x, y)$ be an edge of G, but not in the MST T.
  - What happens, if we add e and remove any other edge, say f, on the path from x to y in T?
  - We will get a MST.
  - But the cost of the MST will be more that T (as $e = (x, y)$ is the heaviest edge in this cycle).

- Can you verify the above for the graph shown in the figure?





MST

- **Lemma:** Let G be a connected, weighted graph. If all edge weights in G are distinct, G has exactly one MST.

- **Proof:**
  - We show that G has exactly one MST by contradiction. Say we have an algorithm that finds an MST (lets call it A). However, assume that MST A is not unique. Then, there must be another spanning tree with equal weight, say MST B.

  - So there must exists an edge, say $e_1 = (u, v)$, which is in A but not in B.
  - But, B (since it's a MST) must have a path from u to v, which can not be the edge e_1.
  - So, if we add e_1 in B, we will get a cycle.

  - Notice that all the edges in this cycle can not be in A (because A is a Tree).
  - So, there must exist an edge in this cycle, say e_2, which belongs to B but not in A.
    - What can we say about the cost of e_1 and e_2?
    - Definitely, $c(e_2) > c(e_1)$… Why? *(Remember, all edge weights are distinct. Read the previous lemma)*
      - e_2 is not in A but if we add e_2 in A, then we will get a cycle and cost of $c(e_2) > c(e_1)$.

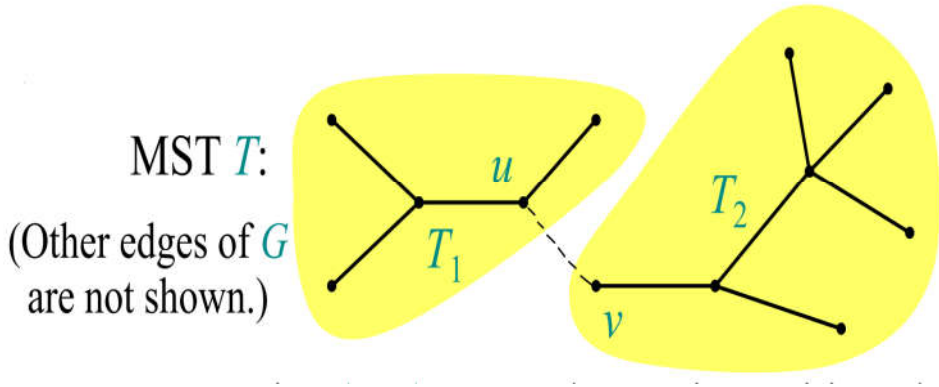
  - So, by replacing e_2 with e_1, we get a spanning tree with total cost less that of B.
  - Which is a contradiction, as we assumed B is a MST.

MST

- Can you state a Naive algorithm for finding MST for a graph… ☺
- What is the complexity of the algorithm?
  - Check, if it is exponential in the worst case!!!
  - Can we say that in the worst case, there can be an exponential number of spanning trees!!!
  - … the answer is YES.

- Thus, we consider Greedy and Dynamic Programming algorithms to solve MST.
- We will see that greedy algorithms can solve MST in nearly linear time.

Optimal substructure for MST



- Consider a MST (T) shown in the figure.
  - Here, T consists of edges of T_1, T_2 and $\{u, v\}$. That is, $T = T_1 \cup T_2 \cup \{u, v\}$
- Remove any edge $(u, v) \in T$. Then, T is partitioned into two subtrees T_1 and T_2 .
- **Lemma-2:** The subtree T_1 is an MST of $G_1 = (V_1, E_1)$, the subgraph of G induced by the vertices of T_1
  - V_1: are the vertices of T_1.
  - E_1: $\{(x, y) \in E : x, y \in V_1\}$. That is, E_1 contain edges where both end vertices are in V_1.
- Similarly for T_2.

Optimal substructure for MST

- **Proof:** (Cut and Paste technique)
- For graph $G = (V, E)$, let us consider the weight of the MST T:

$$w(T) = w(u, v) + w(T_1) + w(T_2)$$

- If T_1 is not a lowest-weight spanning tree for $G_1 = (V_1, E_1)$, then there must exists a MST, say T_1', for G_1.

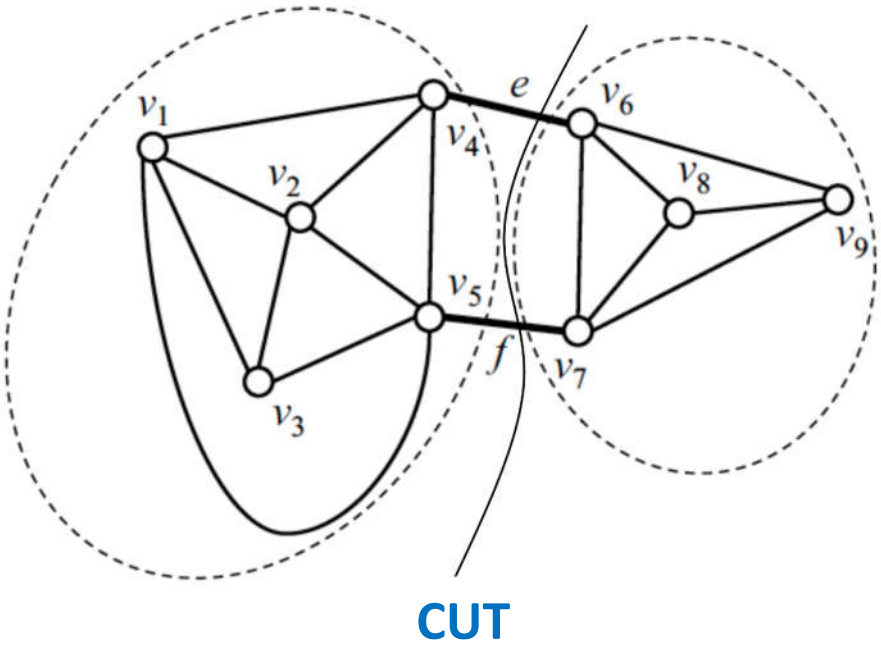
- Then, we can replace T_1 with T_1' to get a spanning tree T' for the whole graph G, which has less weight then T. That is,

$$w(T') = w(u, v) + w(T_1') + w(T_2) < w(T)$$

- This is a contradiction that T is a MST for G.

MST

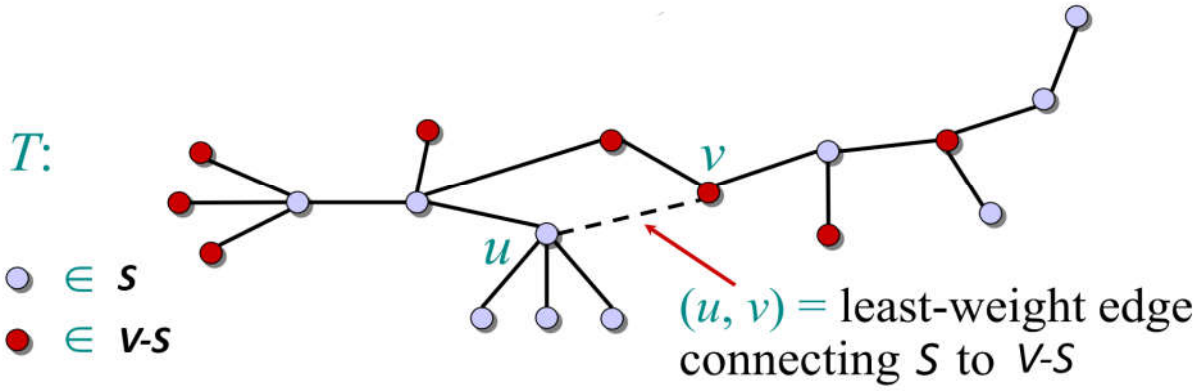
- Similar to cycle property, we will now see the **CUT property**:
- But, before that let's see what is a CUT?
- Let $G = (V, E)$ be a connected and undirected graph. We define:
- **Cut** A cut $(S, V - S)$ of G is a partition of V.
- **Cross** An edge $(u, v) \in \mathrm{E}$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.



MST: Greedy Choice Property

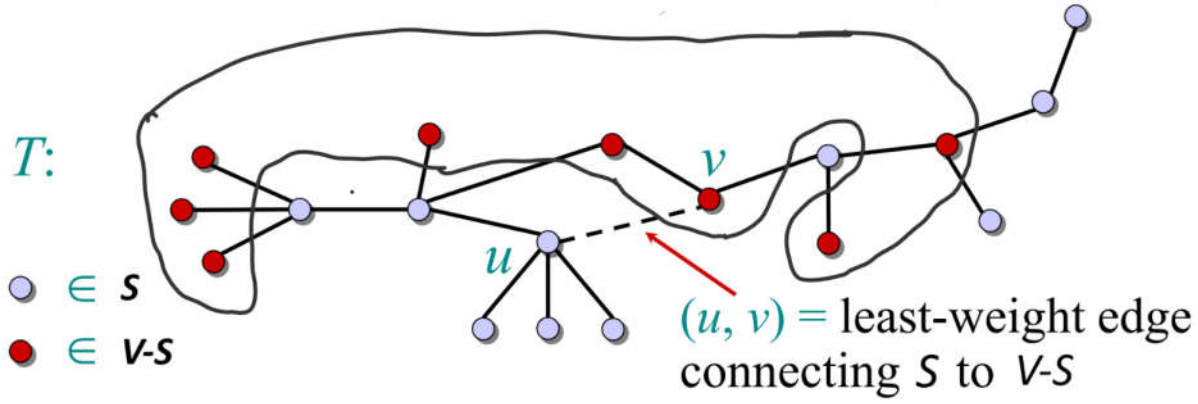
- **Lemma-3:** Let $(S, V - S)$ be a nontrivial cut in G (i.e. $S \neq \emptyset$ and $S \neq V$).
  - If (u, v) is the lowest-cost edge crossing $(S, V - S)$, then (u, v) is in every MST of G.
    - The above is called the CUT property of a graph.
    - And is also the Greedy-Choice Property for a MST.

- Let us consider the following MST, say T, for some graph G. Here we can see, two set of vertices belonging to set S and $V - S$. To be specific, let Grey vertices belong to S and Red belongs to $V - S$.
- Let us assume that we have an edge (u, v) with least weight in graph G, which is not part of the MST T.



- Can you identify S and $V - S$… That is, find the CUT and Cross?
- See Below…

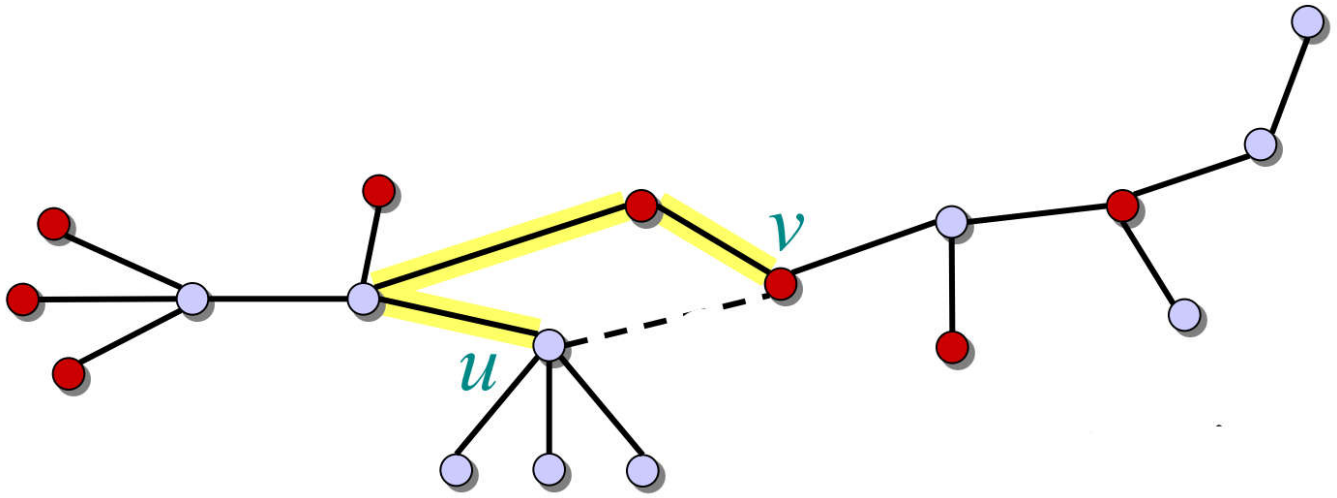
MST: Greedy Choice Property



- Notice that there are many edges in the above figure, that has one end in S and the other in $V - S$, which are called Cross that connects sets S and $V - S$.
- For example, edge (u, v).
- In the previous slide, we assume that $(u, v) \notin T$ has the least weight.

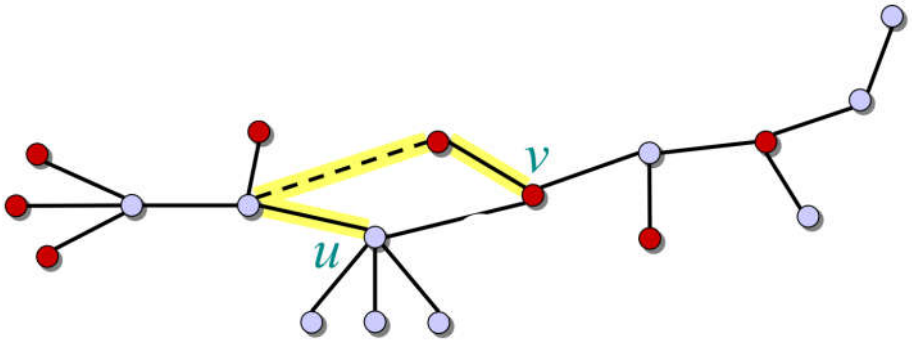
MST: Greedy Choice Property

- Consider the unique simple path from u to v in T.
  - Shown in yellow.
- Swap (u, v) with the first edge on this path that connects a vertex in S to $V - S$.



MST: Greedy Choice Property

- In the figure, we replace an existing edge in MST T (shown in dotted lines) with (u, v).
- Then, what will happen?
  - We will get a lighter-weight spanning tree than T.
  - WHY?



- This is a contradiction that T is the MST for G.

Thus, our assumption that we can have an edge, like (u, v), which is not part of the MST T, is FALSE.

Prim’s Algorithm for MST

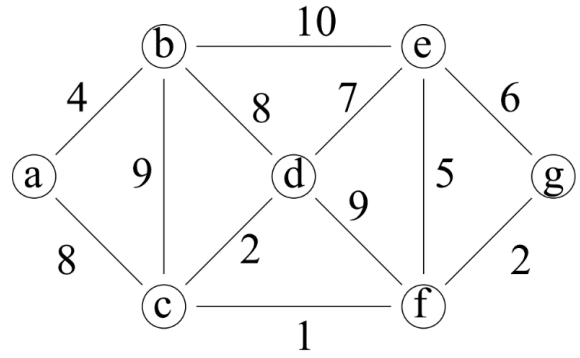
Prim's algorithm

- IDEA: Maintain $V - S$ as a priority queue Q.
  - What is a priority queue?
- The Prim's algorithm makes a nature choice of the cut in each iteration.
  - It grows a single tree and adds a light edge (least weight edge) in each iteration.

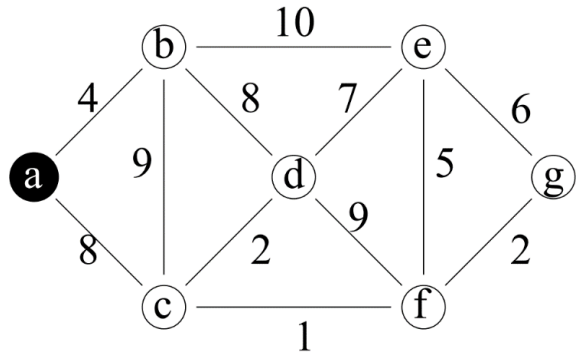
- Let's consider the following example.

Prim's Example

- Consider $G = (V, E)$, shown at the top.
- We want to find the MST for G, using Prim's Algorithm.
- Let's state with vertex 'a'.
  - What will happen, if we start from some other vertex, will we have the same MST?
  - See Lemma-1.



Connected graph



Step 0

S={a}

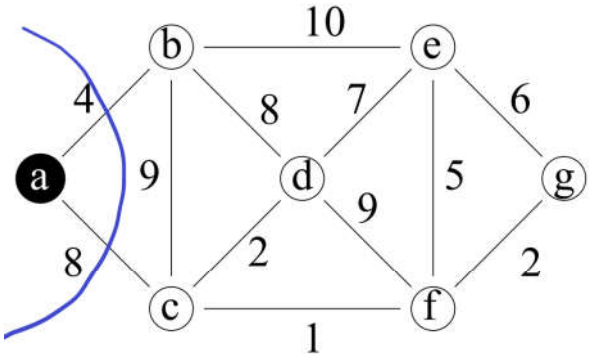
V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

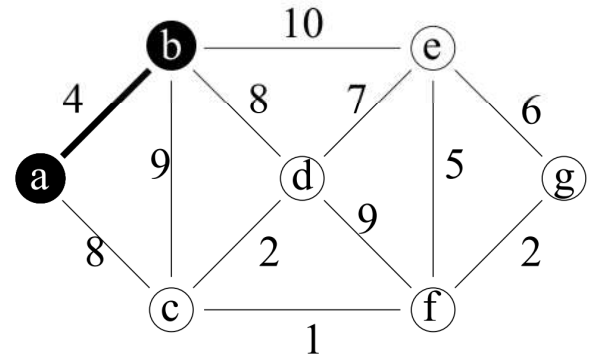
Prim's Example – Continued

CUT

See Lemma-3 for the Greedy choice property… which says, to select an edge (u, v) that is the lowest-cost edge crossing of $(S, V-S)$



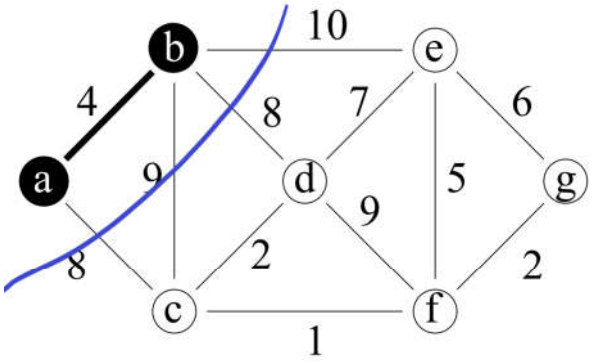
Step 1.1 before
S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}



Step 1.1 after
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

Prim's Example – Continued

CUT



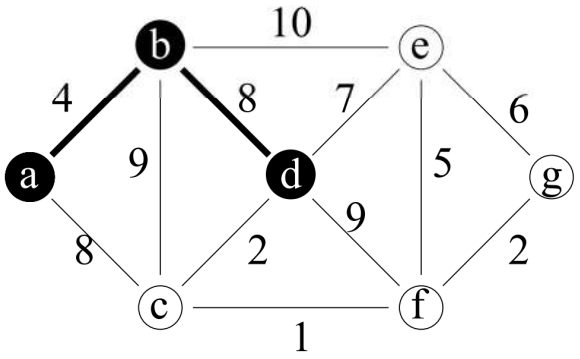
Step 1.2 before

S={a,b}

V \ S = {c,d,e,f,g}

A={{a,b}}

lightest edge = {b,d}, {a,c}



Step 1.2 after

S={a,b,d}

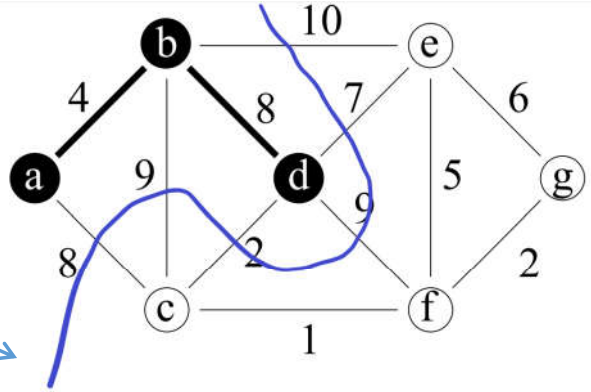
V \ S = {c,e,f,g}

A={{a,b},{b,d}}

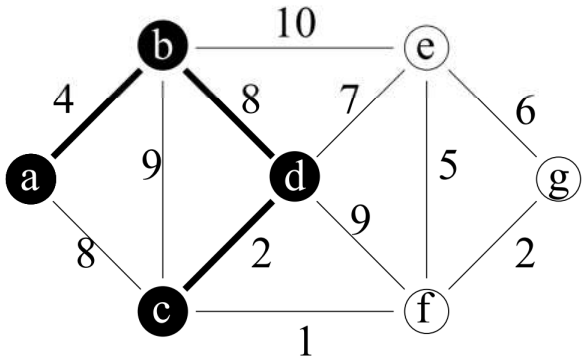
lightest edge = {d,c}

Prim's Example – Continued

CUT

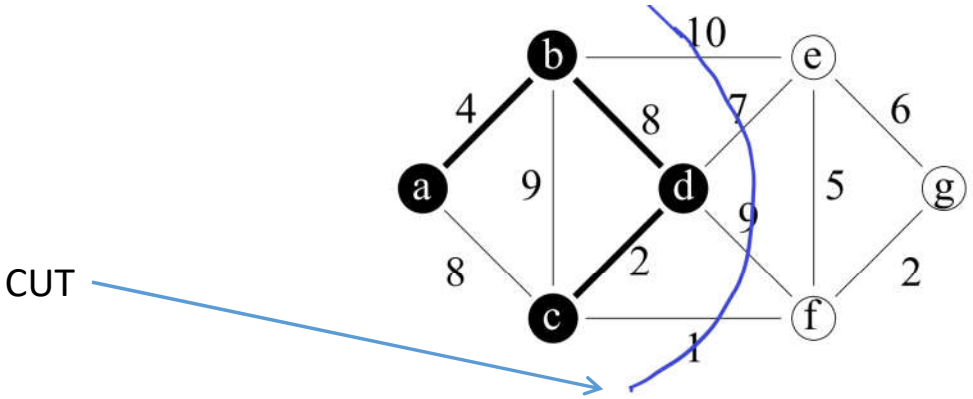


Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

Prim's Example – Continued



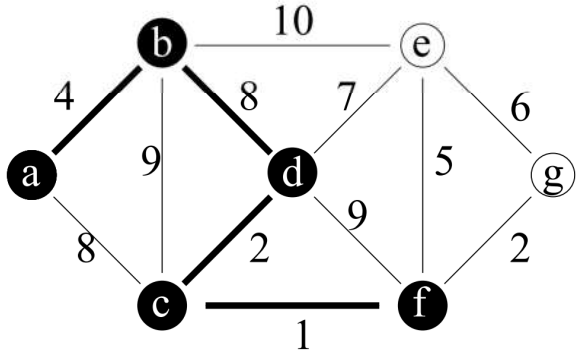
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



Step 1.4 after

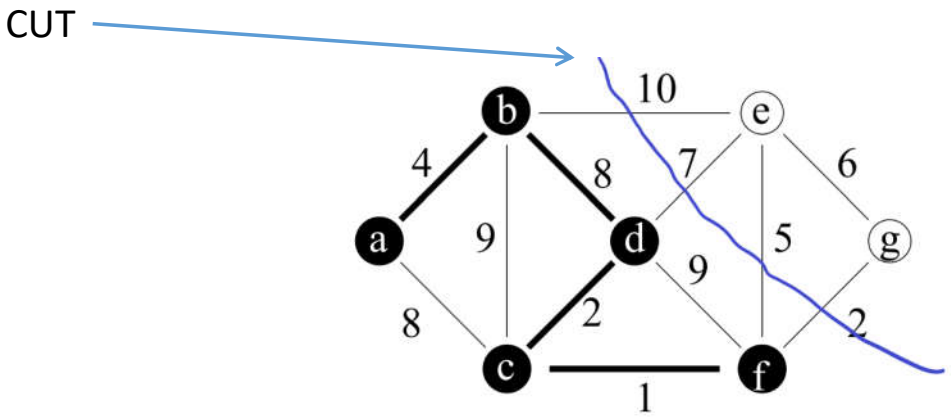
S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}

Prim's Example – Continued



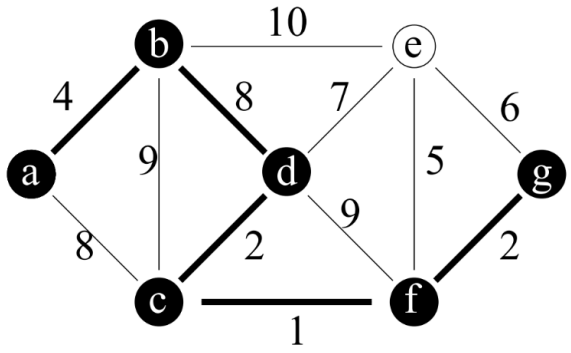
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

S={a,b,c,d,f,g}

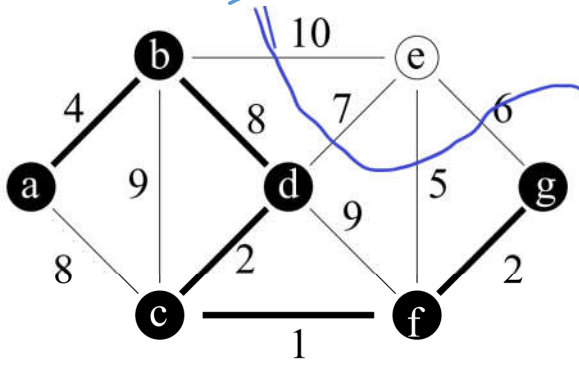
V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

Prim's Example – Continued

CUT



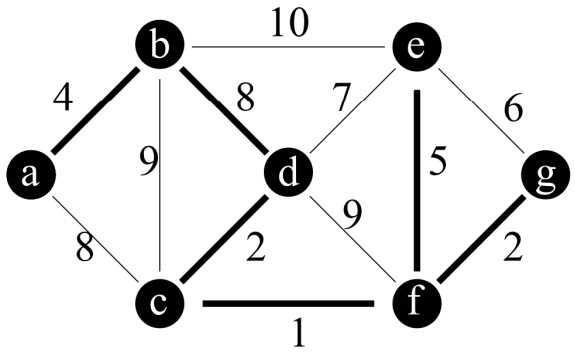
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed

- We will continue the Prim's algorithm in our next slides.

- Homework-1: Try to write the code for the Prim's algorithm
  - What data structure you need.
  - What is the complexity of the algorithm.

- Homework-2: Although we haven't discuss the Prim's and Kruskal's algorithm in detail; but still, can what do you think about the following statement. That is, whether the following statement is TRUE or FALSE?
  - Let G be a connected weighted graph whose edges have distinct weights. Then both Prims' algorithm as well as Kruskal's algorithm produces the same MST.

- Homework-3: Let T_1 and T_2 be two spanning trees of a connected graph G. If edge e is in T_1 but not in T_2, prove that there exists another edge f in T_2 but not in T_1 such that $(T_1 - e) \cup f$ and $(T2 - f) \cup e$ are also spanning trees of G.
  - Hint: Notice, adding f in T_1 creates a cycle… and removing e from T_1 breaks the cycle, isn't it?
    - If so then, $(T_1 - e) \cup f$ is a spanning tree.
    - However, you may need to think about the 1st statement.

- Homework-4:
  - Prove that, if e be the only minimum cost edge of G, then e belongs to every MST of G.

- Homework-5: Describe five applications of MST.

Greedy Algorithm

Graph Problems: Minimum Spanning Trees

The content of this presentation is a collection from various books and online resources, such as Introduction to Algorithms, by CLRS and the lecture notes of Dr. K. Schwarz and others. Thanks to all for their valuable contributions.

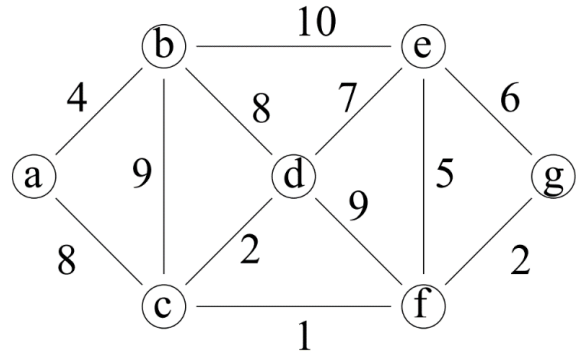
Prim's algorithm (Recall)

- IDEA: Maintain $V - S$ as a priority queue Q.
  - What is a priority queue?
- The Prim's algorithm makes a nature choice of the cut in each iteration.
  - It grows a single tree and adds a light edge (least weight edge) in each iteration.

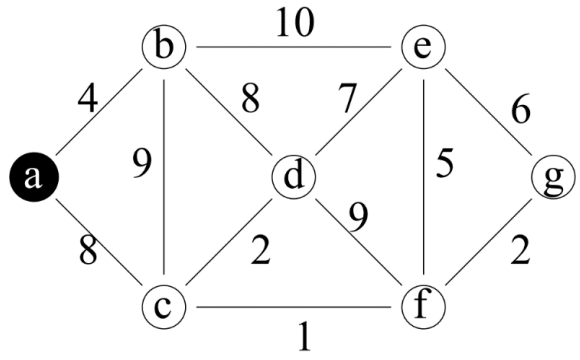
- Let's consider the following example.

Prim's Example

- Consider $G = (V, E)$, shown at the top.
- We want to find the MST for G, using Prim's Algorithm.
- Let's state with vertex 'a'.
  - What will happen, if we start from some other vertex, will we have the same MST?
  - See Lemma-1.



Connected graph



Step 0

S={a}

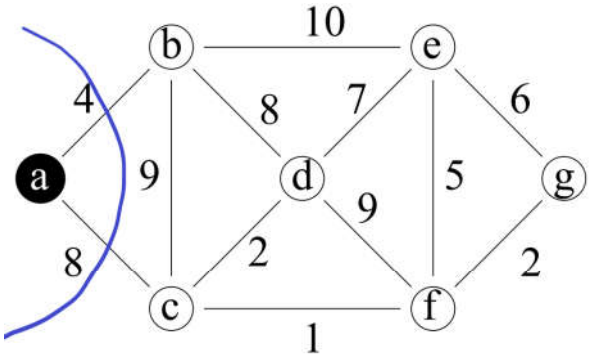
V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

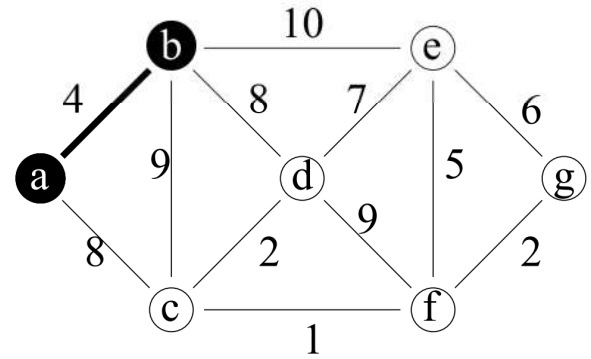
Prim's Example – Continued

CUT

See Lemma-3 for the Greedy choice property… which says, to select an edge (u, v) that is the lowest-cost edge crossing of $(S, V-S)$



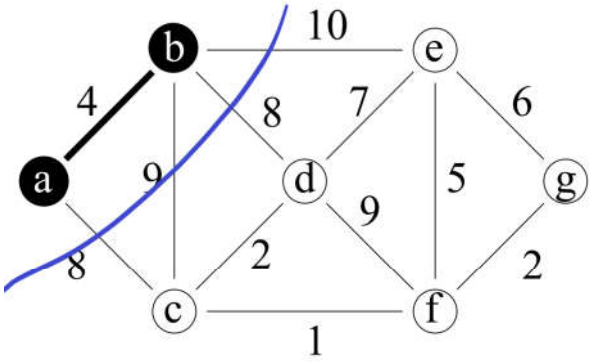
Step 1.1 before
S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}



Step 1.1 after
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

Prim's Example – Continued

CUT



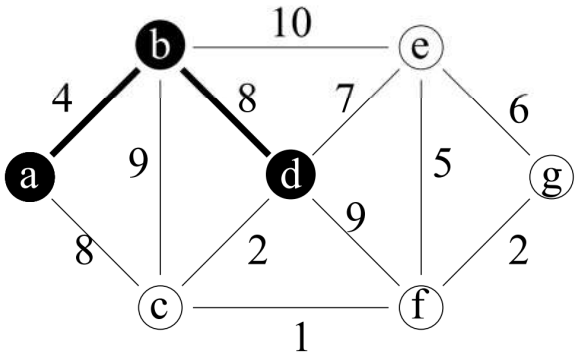
Step 1.2 before

S={a,b}

V \ S = {c,d,e,f,g}

A={{a,b}}

lightest edge = {b,d}, {a,c}



Step 1.2 after

S={a,b,d}

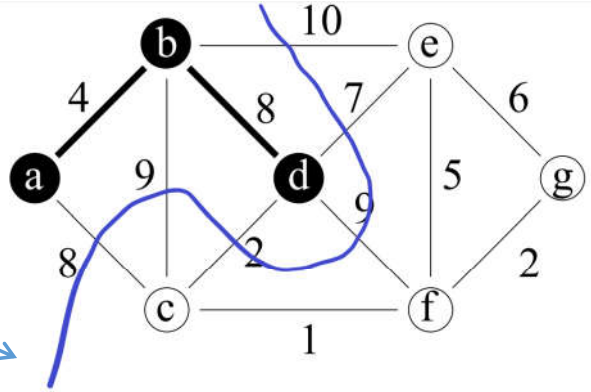
V \ S = {c,e,f,g}

A={{a,b},{b,d}}

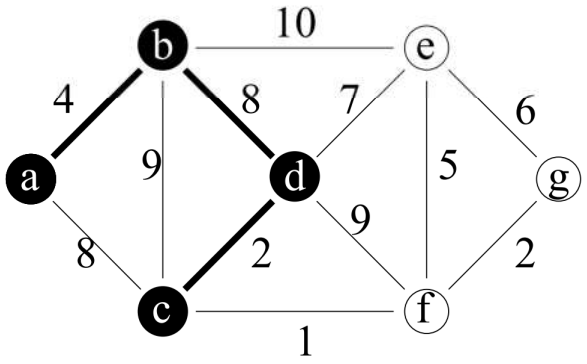
lightest edge = {d,c}

Prim's Example – Continued

CUT

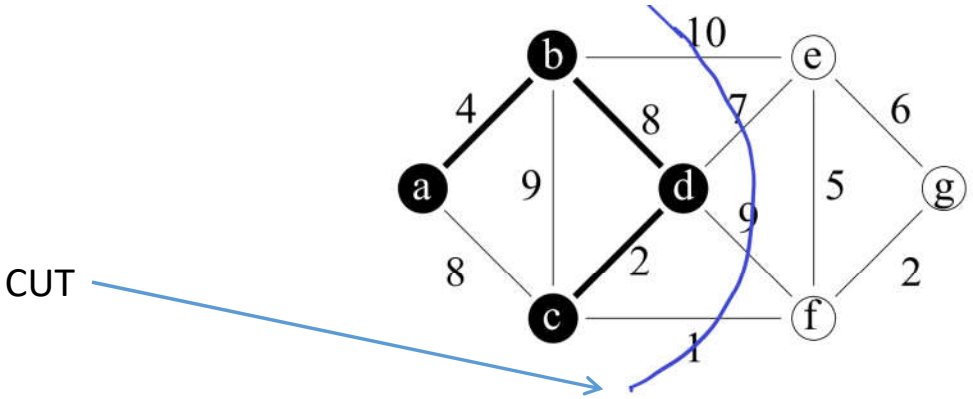


Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

Prim's Example – Continued



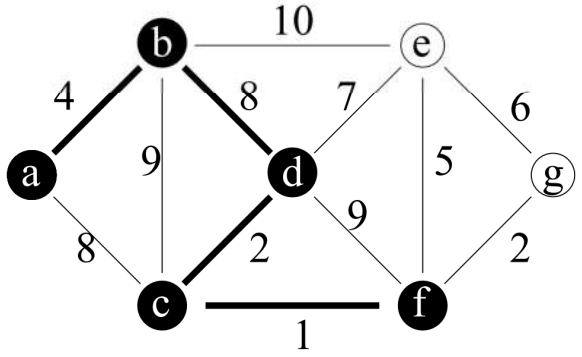
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



Step 1.4 after

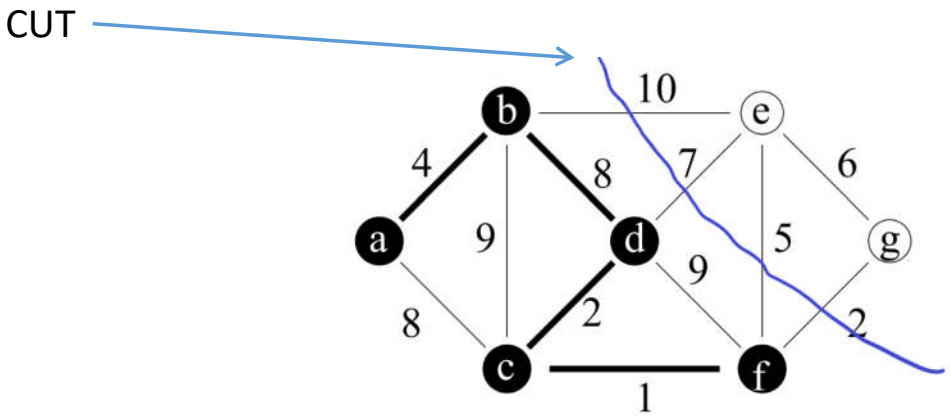
S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}

Prim's Example – Continued



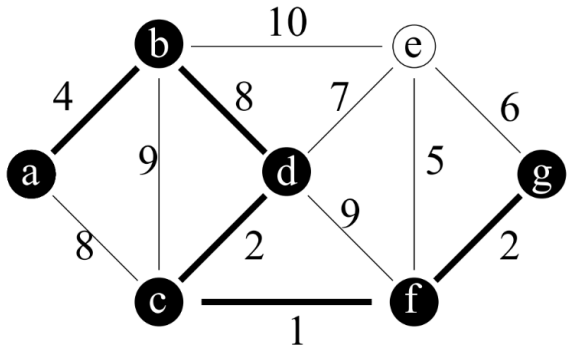
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

S={a,b,c,d,f,g}

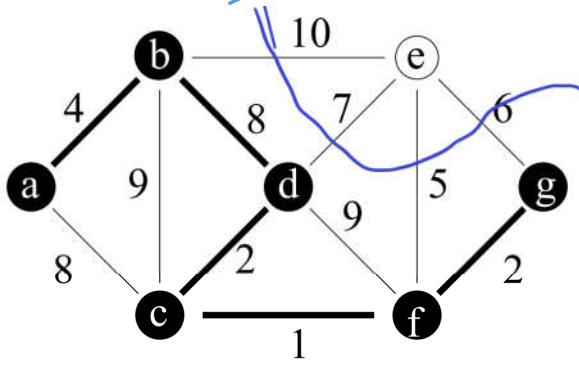
V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

Prim's Example – Continued

CUT



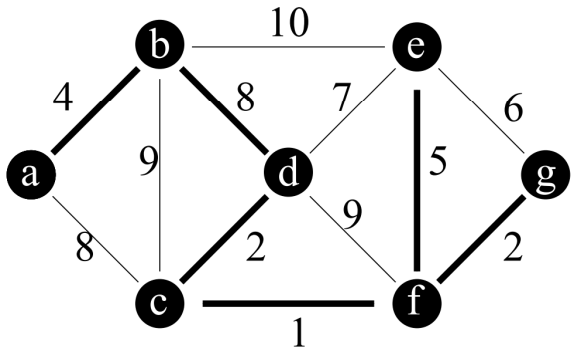
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed

Prim's Algorithm

- From the implementation point… here an the algorithm.

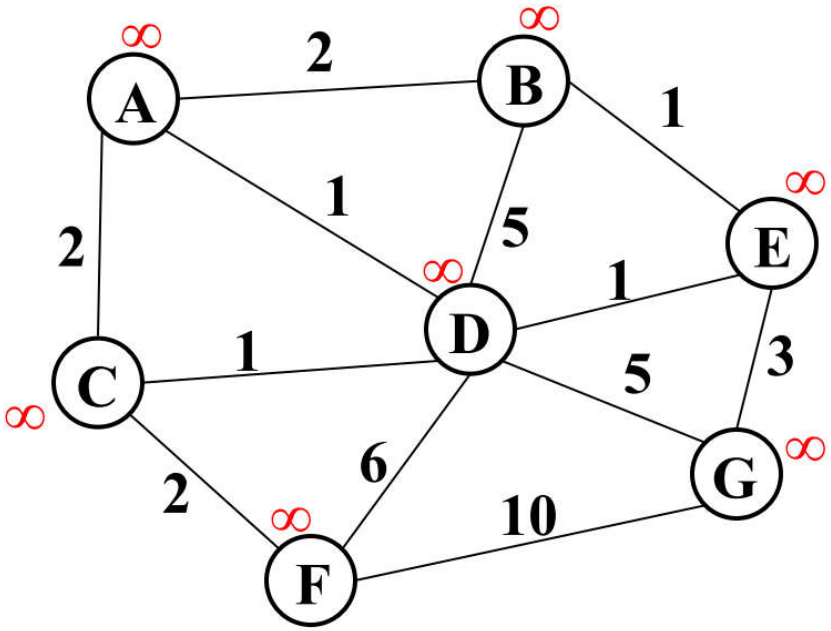
- During execution of the algorithm, all vertices that are *not* in the tree reside in a min-priority queue Q based on a *key* attribute.
  - We may consider a binary Heap data structure for the same.

- Apart from a heap, we need
  - Adjacency matrix or list for storing graph $G = (V, E)$.
  - An array $v.key$ to mark the cost required by node v, for joining the tree.
  - An array, $v.\pi$, to note the parents.
  - An array, $v.color$ to mark the visited nodes by the algorithm.
  - Another array, $v.heap$ to keep the index of the vertices in the Heap.

MST-PRIM(G, w, r)

```
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

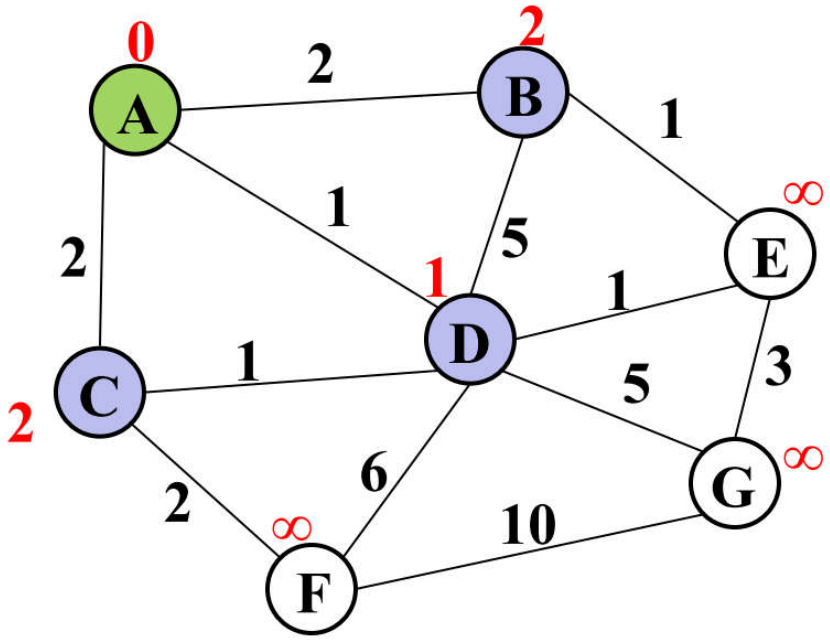
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	White	∞	NIL
B	White	∞	NIL
C	White	∞	NIL
D	White	∞	NIL
E	White	∞	NIL
F	White	∞	NIL
G	White	∞	NIL

- Let's start from vertex A. Mark $A.key = 0$, and for all other vertices $v.key = \infty$
- Put all keys in the priority queue Q.
- Start by extracting the minimum, i.e., of course node A

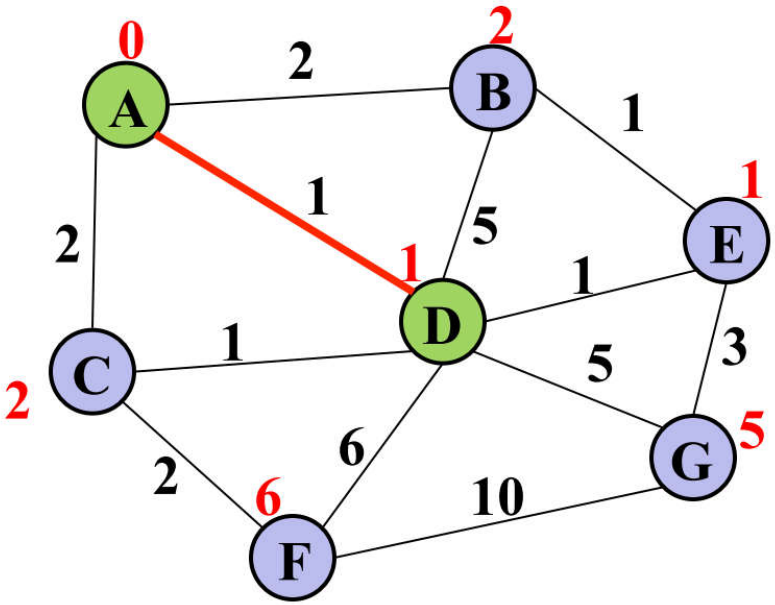
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B		2	A
C		2	A
D		1	A
E		∞	NIL
F		∞	NIL
G		∞	NIL

- In the 1st iteration, Extract-Min will returns A
- Updates the data structures, as required.

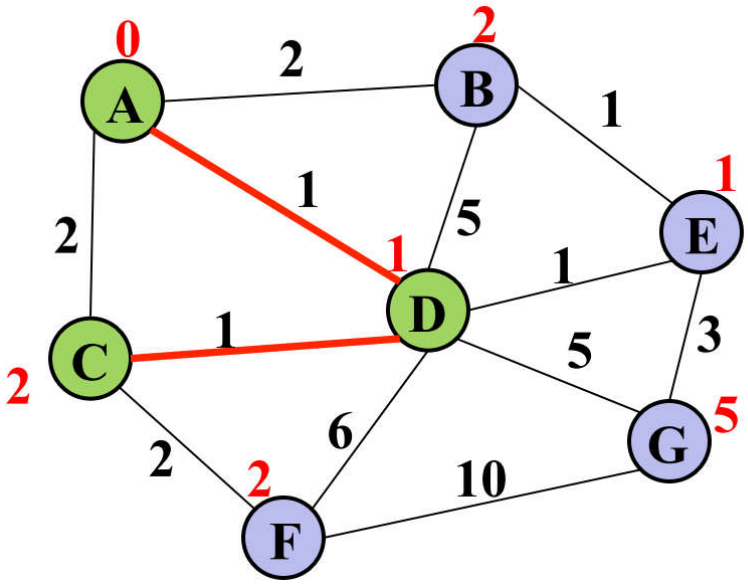
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B		2	A
C		1	D
D	Green	1	A
E		1	D
F		6	D
G		5	D

- In the next iteration, Extract-Min will return D
- Check all the edges passing from D, and update the data structure.
- Note that in the above table, we need to update node C, as well. Because $w(C, D) < C.key$.

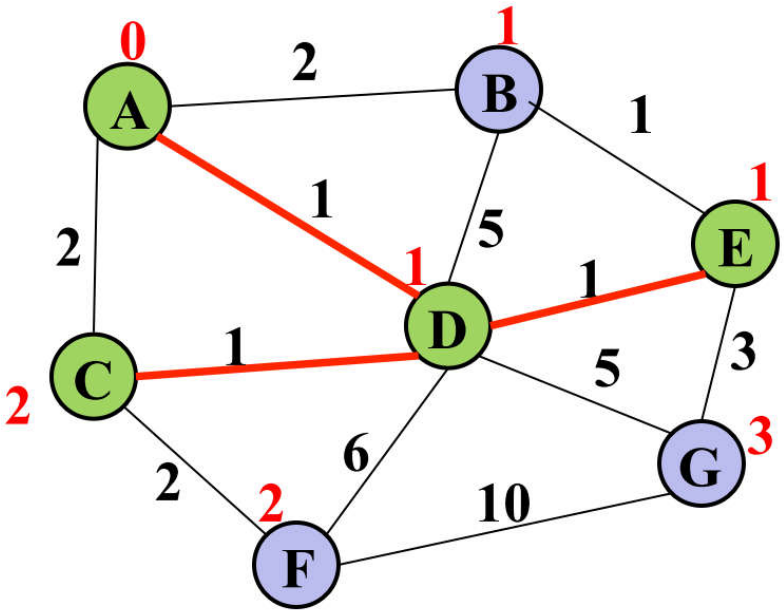
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B		2	A
C	Green	1	D
D	Green	1	A
E		1	D
F		2	C
G		5	D

- In the next iteration, Extract-Min will return C
- Check all the edges passing from C, and update the data structure.
- Note that in the above table, we need to update node F, as well. Because $w(F, C) < F.key$.

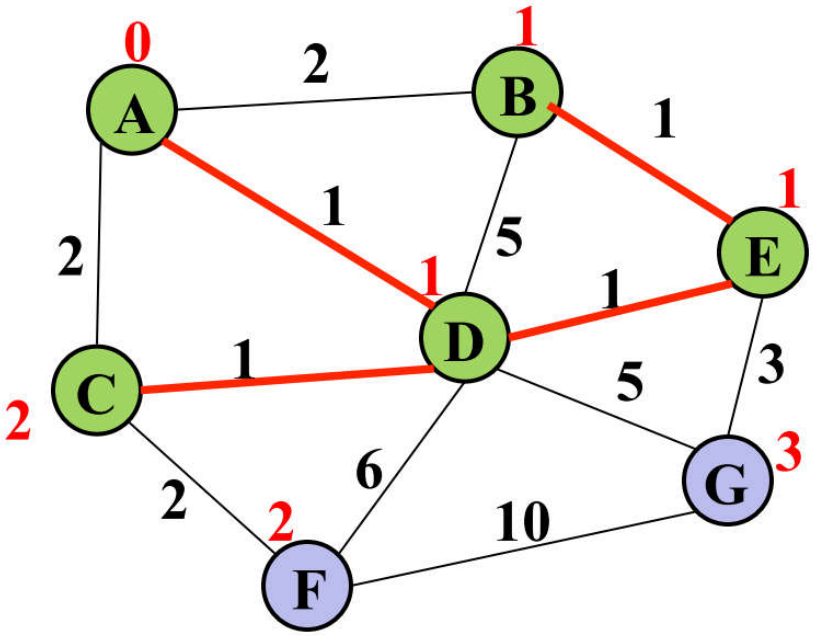
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B		1	E
C	Green	1	D
D	Green	1	A
E	Green	1	D
F		2	C
G		3	E

- In the next iteration, Extract-Min will return E
- Check all the edges passing from E, and update the data structure.
- Note that in the above table, we need to update node F, as well. Because $w(F, C) < F.key$.

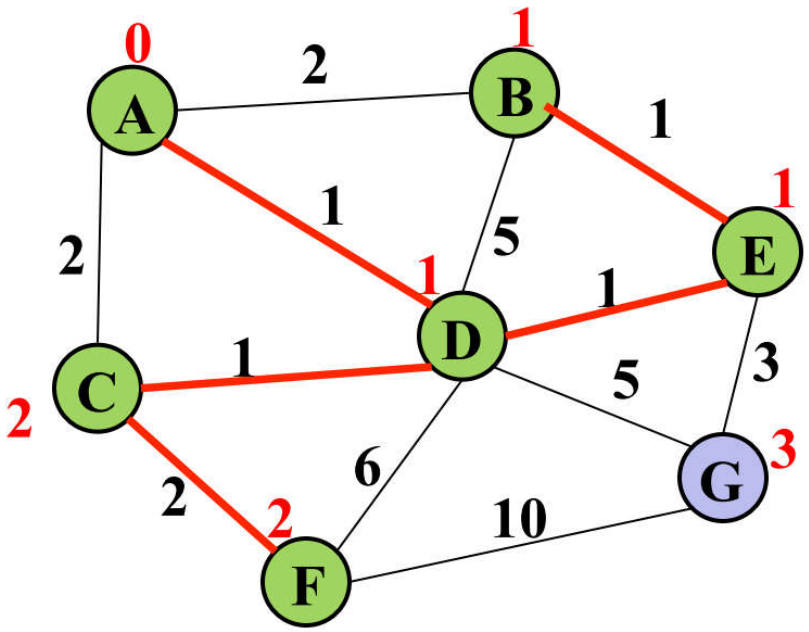
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B	Green	1	E
C	Green	1	D
D	Green	1	A
E	Green	1	D
F		2	C
G		3	E

- In the next iteration, Extract-Min will return B
- Check all the edges passing from B, and update the data structure.
- No change in $v.key$

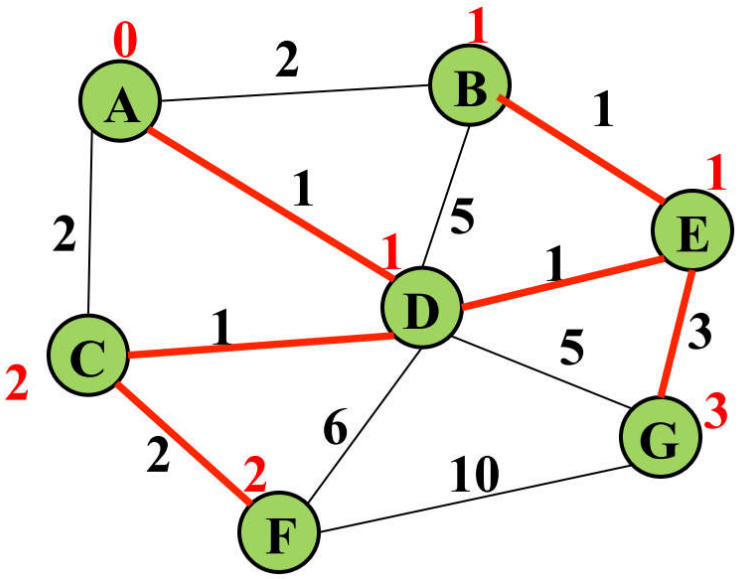
Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B	Green	1	E
C	Green	1	D
D	Green	1	A
E	Green	1	D
F	Green	2	C
G		3	E

- In the next iteration, Extract-Min will return F
- Check all the edges passing from F, and update the data structure.
- No change in $v.key$.

Prim's Algorithm (Example)



	$v.color$	$v.key$	$v.\pi$
A	Green	0	A
B	Green	1	E
C	Green	1	D
D	Green	1	A
E	Green	1	D
F	Green	2	C
G	Green	3	E

- In the next iteration, Extract-Min will return G
- Check all the edges passing from G, and update the data structure.
- No change in $v.key$
- The final MST is shown above.

Prim's Algorithm - Complexity

- Finally, what is the complexity of the Prim's algorithm?

- Initialization in the first few lines 1-5 is $O(V)$.
- Complexity of a single **Extract-Min** operation on a **Binary Min-Heap** is $O(\log V)$. The while loop executes the Extract-Min operation for each V. So the complexity of Line-7 is O(V log V).
- In line-8, the for-loop is executed $2E$ times, in total. (WHY?).
  - The time required to execute line 11 is O(log V), by using the **Decrease_Key** operation on the min_heap.
  - Thus the total complexity is $O(E \log V)$.

- Thus, the total complexity of the Prim's algorithm is $O(V \log V + E \log V) = O\big((E + V) \log V\big) = O(E \log V)$.

- However, if you use **Fibonacci Heap**, we can have a complexity of $O(E + V \log V)$.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

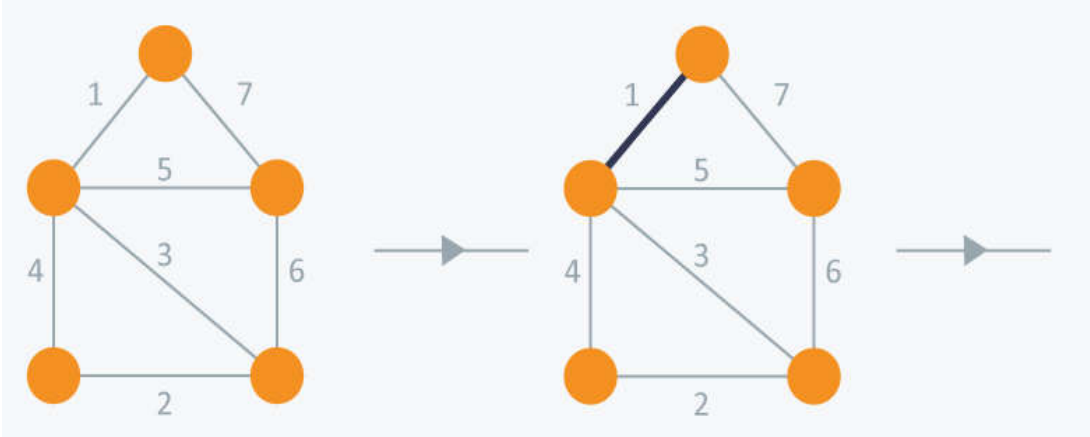
Kruskal’s Algorithm

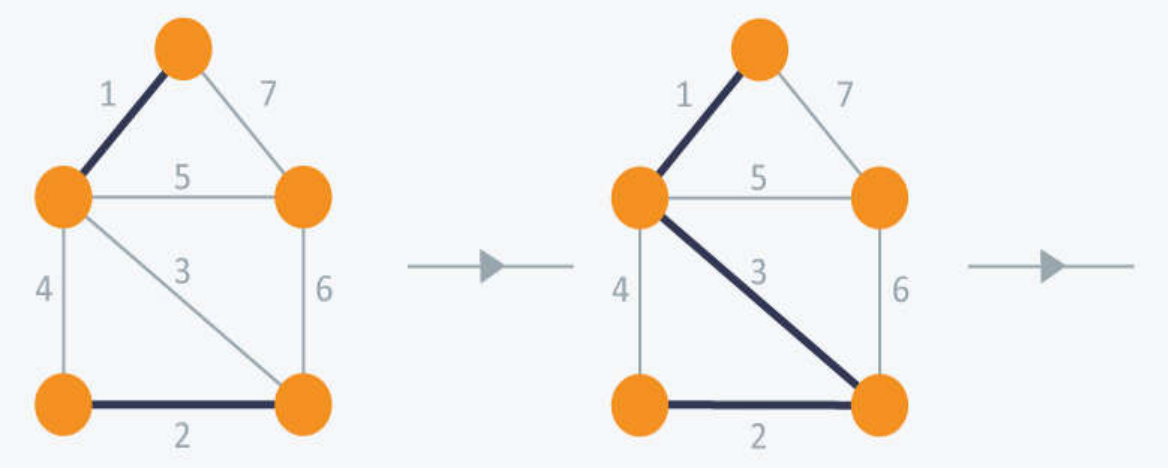
Kruskal's Algorithm

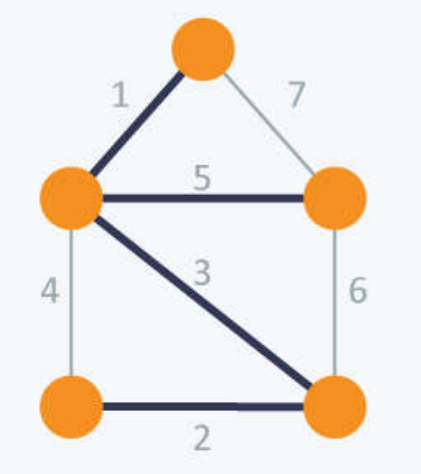
- Kruskal's algorithm. [Kruskal, 1956]
  - Algorithm starts with the all the nodes and no edges, so there is a forest of trees, each of which is a single node (a leaf).
  - Consider edges in ascending order of cost.
  - Add an edge with minimum weight to T unless doing so would create a cycle.
  - Can have a forest of trees.

- Unlike Prim's algorithm, we make a different choices of cuts in the Kruskal's algorithm.
- Moreover, observe that unlike Prim's algorithm, which only grows one tree, Kruskal's algorithm grows a collection of trees (a forest).
- The algorithm continues until the forest is 'merged to' a single tree.

Kruskal's Algorithm







Kruskal's Algorithm

- **Question:** How would we check if adding an edge $\{u, v\}$ would create a cycle?

- *Observation:* If u and v are in the same tree (in a forest), then adding edge $\{u, v\}$ to creates a cycle.
  - What will happen if nodes u and v are in two different trees, say T_1 and T_2, and suppose that the next edge in the sorted order is $\{u, v\}$?
  - In that case, adding $\{u, v\}$ will not create any cycle
  - We need to merge T_1 and T_2.
- However, if both u and v are in the same tree T, adding $\{u, v\}$ will creates a cycle.

Kruskal's Algorithm

- **Question:** How to test whether u and v are in the same tree or not?
- Answer: Use a *disjoint-set data structure.*
  - The UNION-FIND data structure implements this.
  - UNION-FIND supports three operations on collections of disjoint sets: Let n be the size of the universe (total number of vertices, in our case).
  - Create-Set(u): $O(1)$.
    - Create a set containing the single element u.
  - Find-Set(u): $O(\log n)$
    - Find the set containing the elements u.
  - Union(u, v): $O(\log n)$
    - Merge the sets respectively containing u and v into a common set.

Kruskal's Algorithm: the Details

```
Sort E in increasing order by weight w;        O(|E| log |E|)
/* After sorting E = ⟨{u_1, v_1}, {u_2, v_2}, ..., {u_|E|, v_|E|}⟩ */

A = { };
for (each u in V) CREATE-SET(u);               O(|V|)

for e_i = (u_i, v_i) from 1 to |E| do          O(|E| log |V|)
    if (FIND-SET(u_i) != FIND-SET(v_i) )
    {   add {u_i, v_i} to A;
        UNION(u_i, v_i);
    }
return(A);
```

Remark: With a proper implementation of UNION-FIND, Kruskal's algorithm has running time $O(E \log E)$.

Correctness Proof Intuition

- **Theorem:** Kruskal's algorithm always produces an MST.
- **Proof:**
- Let T be the tree produced by the Kruskal's algorithm and T* be a MST. We will prove that the cost of both trees are same, i.e., c(T) = c(T*).
- Firstly, if T = T*, we are done.
- Otherwise, lets assume that T ≠ T*, so T–T* ≠ ∅. In other words, there is an edge in T with is not in T*. Let's assume that (u, v) be an edge in T–T*.
- Let S be the sub-set (or sub tree) containing u at the time (u, v) was added by the Kruskal algorithm to T.
- We claim that the (u, v) is a least-cost edge crossing cut $(S, V - S)$. First, (u, v) crosses the cut, since u and v were not connected when Kruskal's algorithm selected (u, v).
- Next, if there were a lower-cost edge e crossing the cut, e would connect two nodes that were not connected. Thus, Kruskal's algorithm would have selected e instead of (u, v), a contradiction.

Correctness Proof Intuition

- Since T* is an MST, there is a path from u to v in T*.
- The path begins in S and ends in $V - S$, so it contains an edge (x, y) crossing the cut.
- Then T*' = T* ∪ {(u, v)} – {(x, y)} is a spanning tree of G and c(T*') = c(T*) + c(u, v) – c(x, y).
- Since c(x, y) ≥ c(u, v), we have c(T*') ≤ c(T*).
- Since T* is an MST, c(T*') = c(T*)
- Therefore, if we repeat this process once for each edge in T – T*, we will have converted T* into T while preserving c(T*).
- Thus c(T) = c(T*).